This International Student Edition is for use outside of the U.S.

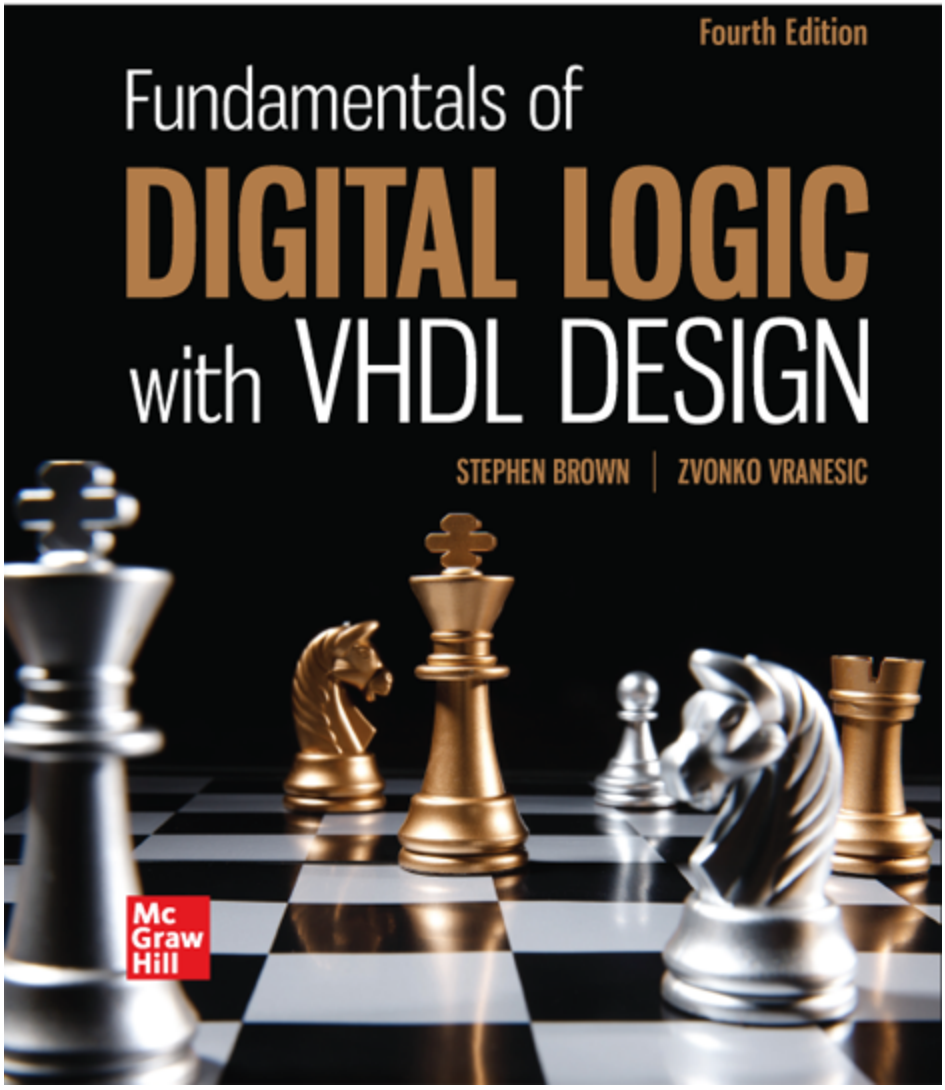
Fourth Edition

Fundamentals of DIGITAL LOGIC with VHDL DESIGN

STEPHEN BROWN | ZVONKO VRANESIC

Mc Graw Hill

This International Student Edition is for use outside of the U.S.



FUNDAMENTALS OF DIGITAL LOGIC WITH VHDL DESIGN

FOURTH EDITION

Stephen Brown and Zvonko Vranesic

Department of Electrical and Computer Engineering
University of Toronto

Mc
Graw
Hill

FUNDAMENTALS OF DIGITAL LOGIC WITH VHDL DESIGN

Published by McGraw Hill LLC, 1325 Avenue of the Americas, New York, NY 10019. Copyright c © 2023 by McGraw Hill LLC. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of McGraw Hill LLC, including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 LCR 27 26 25 24 23 22

ISBN 978-1-260-59778-3
MHID 1-260-59778-4

Cover Image: *View Stock/Getty Images*

All credits appearing on page or at the end of the book are considered to be an extension of the copyright page.

The Internet addresses listed in the text were accurate at the time of publication. The inclusion of a website does not indicate an endorsement by the authors or McGraw Hill LLC, and McGraw Hill LLC does not guarantee the accuracy of the information presented at these sites.

mheducation.com/highered

To Susan and Anne

ABOUT THE AUTHORS

Stephen Brown received the Ph.D. and M.A.Sc. degrees in Electrical Engineering from the University of Toronto, and his B.A.Sc. degree in Electrical Engineering from the University of New Brunswick. He joined the University of Toronto faculty in 1992, where he is now a Professor in the Department of Electrical & Computer Engineering. He is also the Director of FPGA Academic Programs for Intel Corporation.

His research interests include field-programmable VLSI technology, CAD algorithms, computer architecture, and applications of machine learning. He won the Canadian Natural Sciences and Engineering Research Council's 1992 Doctoral Prize for the best Ph.D. thesis in Canada, and the New Brunswick Governor-General's 1985 award for the highest academic standing in the Faculty of Engineering. He is a coauthor of more than 150 scientific research papers and two other textbooks: *Fundamentals of Digital Logic with Verilog Design* and *Field-Programmable Gate Arrays.*

He has won many awards for excellence in teaching electrical engineering, computer engineering, and computer science courses.

Zvonko Vranesic received his B.A.Sc., M.A.Sc., and Ph.D. degrees, all in Electrical Engineering, from the University of Toronto. From 1963 to 1965 he worked as a design engineer with the Northern Electric Co. Ltd. in Bramalea, Ontario. In 1968 he joined the University of Toronto, where he is now a Professor Emeritus in the Department of Electrical & Computer Engineering. During the 1978-79 academic year, he was a Senior Visitor at the University of Cambridge, England, and during 1984-85 he was at the University of Paris, 6. From 1995 to 2000 he served as Chair of the Division of Engineering Science at the University of Toronto.

He is a coauthor of four other books: *Computer Organization and Embedded Systems*, 6th ed.; *Fundamentals of Digital Logic with Verilog Design*, 3rd ed.; *Microcomputer Structures;* and *Field-Programmable Gate Arrays.* In 1990, he received the Wighton Fellowship for "innovative and distinctive contributions to undergraduate laboratory instruction." In 2004, he received the Faculty Teaching Award from the Faculty of Applied Science and Engineering at the University of Toronto.

He has represented Canada in numerous chess competitions. He holds the title of International Master.

PREFACE

This book is intended for an introductory course in digital logic design, which is a basic course in most electrical and computer engineering programs. A successful designer of digital logic circuits needs a good understanding of the classical methods of logic design and a firm grasp of the modern design approach that relies on computer-aided design (CAD) tools.

The main goals of this book are (1) to teach students the fundamental concepts of classical manual digital design and (2) illustrate clearly the way in which digital circuits are designed today, using CAD tools. Even though current modern designers very rarely use manual techniques, our motivation for teaching such techniques is to give students an intuitive feeling for how digital circuits operate. The manual techniques also provide an illustration of the types of manipulations performed by CAD tools, giving students an appreciation of the benefits provided by design automation. Throughout the book, basic concepts are introduced by way of simple design examples, which we perform using both manual techniques and modern CAD-tool-based methods. After basic concepts are established, more complex examples are provided using the CAD tools. Thus, to illustrate how digital design is presently carried out in practice, our emphasis is on modern design methodology.

TECHNOLOGY

This book discusses modern digital circuit implementation technologies. The emphasis is placed on programmable logic devices (PLDs), which is the most appropriate technology for use in a textbook for two reasons. First, PLDs are widely used in practice and are suitable for almost all types of digital circuit designs. In fact, students are more likely to be involved in

PLD-based designs at some point in their careers than in any other technology. Second, circuits are implemented in PLDs by end-user programming. Therefore, students can be provided with the opportunity, in a laboratory setting, to implement this book's design examples in actual chips. Students can also simulate the behavior of their designed circuits on their own computers. We use the two most popular types of PLDs for targeting of designs: complex programmable logic devices (CPLDs) and field-programmable gate arrays (FPGAs).

We emphasize the use of a hardware description language (HDL) in specifying the logic circuits, because an HDL-based approach is the most efficient design method to use in practice. We describe in detail the IEEE Standard VHDL language and use it extensively in examples.

SCOPE OF THE BOOK

This edition of the book has been extensively restructured. All of the material that should be covered in a one-semester course is now included in Chapters 1 to 6. More advanced material is presented in Chapters 7 to 11.

Chapter 1 provides a general introduction to the process of designing digital systems. It discusses the key steps in the design process and explains how CAD tools can be used to automate many of the required tasks. It also introduces the representation of digital information.

Chapter 2 introduces logic circuits. It shows how Boolean algebra is used to represent such circuits. It introduces the concepts of logic circuit synthesis and optimization, and shows how logic gates are used to implement simple circuits. It also gives the reader a first glimpse at VHDL, as an example of a hardware description language that may be used to specify the logic circuits.

Chapter 3 concentrates on circuits that perform arithmetic operations. It discusses numbers and shows how they can be manipulated using logic circuits. This chapter illustrates how VHDL can be used to specify a desired functionality and how CAD tools can provide a mechanism for developing the required circuits.

Chapter 4 presents combinational circuits such as multiplexers, encoders, and decoders that are used as building blocks in larger circuits. These circuits are very convenient for illustrating the application of many VHDL

constructs, giving the reader the opportunity to discover more advanced features of VHDL.

Chapter 5 introduces storage elements. The use of flip-flops to realize regular structures, such as shift registers and counters, is discussed. VHDL-specified designs of these structures are included.

Chapter 6 gives a detailed presentation of synchronous sequential circuits (finite state machines). It explains the behavior of these circuits and develops practical techniques for both manual and automated design.

Chapter 7 is a discussion of a number of issues that arise in the design of digital logic systems. It reveals and highlights problems that are often encountered in practice, and explains how they can be overcome. Examples of larger circuits illustrate a hierarchical approach in designing digital systems. Complete VHDL code for these circuits is presented.

Chapter 8 deals with more advanced methods for optimized implementation of logic functions. It presents algorithmic techniques for optimization, and also explains how logic functions can be specified using a cubical representation or binary decision diagrams.

Chapter 9 discusses asynchronous sequential circuits. While this treatment is not exhaustive, a clear indication of the main characteristics of such circuits is provided. Even though asynchronous circuits are not used extensively in practice, they provide an excellent vehicle for gaining a more complete understanding of the operation of digital circuits. Asynchronous circuits illustrate the importance of clearly understanding how the structure of a circuit affects the propagation delays of signals, and how these delays can impact a circuit's functionality.

Chapter 10 presents a complete CAD flow that a designer experiences when creating, implementing, and testing a digital circuit.

Chapter 11 introduces the topic of testing. A designer of logic circuits must be aware of the need to test circuits and should be conversant with at least the most basic aspects of testing.

Appendix A provides a complete summary of VHDL features. Although use of VHDL is integrated throughout the book, this appendix provides a convenient reference that the reader may occasionally consult while writing VHDL code.

Appendix B presents the electronic aspects of digital circuits. This appendix shows how the basic gates are built using transistors and presents various factors that affect circuit performance. The most modern

technologies are emphasized, with particular focus on CMOS technology and PLDs.

WHAT CAN BE COVERED IN A COURSE

Much of the material in the book can be covered in 2 one-quarter courses. Thorough coverage of the most important material can be achieved in a single one-semester, or even a one-quarter, course. This is possible only if the instructor does not spend too much time discussing the intricacies of VHDL and CAD tools. To make this approach possible, we organized the VHDL material in a modular style that is conducive to self-study. Our experience in teaching students at the University of Toronto has shown that the instructor may only spend a total of three to four lecture hours on VHDL, describing how the code should be structured (including the use of design hierarchy) using scalar and vector variables, and the style of code needed to specify sequential circuits. The VHDL examples given in this book are largely self-explanatory, and students can understand them with ease.

The book is also suitable for a course in logic design that does not include exposure to VHDL. However, some knowledge of VHDL, even at a rudimentary level, is beneficial to the students, and is great preparation for a job as a design engineer.

One-Semester Course

The following material should be covered in lectures:

- Chapter 1—all sections.
- Chapter 2—all sections.
- Chapter 3—sections 3.1 to 3.5.
- Chapter 4—all sections.
- Chapter 5—all sections.
- Chapter 6—all sections.

One-Quarter Course

In a one-quarter course the following material can be covered:

- Chapter 1—all sections.
- Chapter 2—all sections.
- Chapter 3—sections 3.1 to 3.3 and section 3.5.
- Chapter 4—all sections.
- Chapter 5—all sections.
- Chapter 6—sections 6.1 to 6.4.

VHDL

VHDL is a complex language that some instructors feel is too hard for beginning students to grasp. We fully appreciate this issue and have attempted to solve it by presenting only the most important VHDL constructs that are useful for the design and synthesis of logic circuits. Many other language constructs, such as those that have meaning only when using the language for simulation purposes, are omitted. The VHDL material is integrated gradually, with more advanced features presented only at points where their use can be demonstrated in the design of relevant circuits.

The book includes more than 150 examples of VHDL code. These examples illustrate how VHDL is used to describe a wide range of logic circuits, from those that contain only a few gates to those that represent digital systems such as a simple processor.

CAD Tools

All of the examples of VHDL code presented in the book are provided on the authors' website at

www.eecg.toronto.edu/~brown

To gain a more thorough understanding of the VHDL code examples, the reader is encouraged to compile and simulate the code using a commercially-available simulation CAD tool. Toward this end each example of VHDL code on the Authors' website is accompanied by setup files for the widely-used *ModelSim* simulator available from Mentor

Graphics. A detailed tutorial is provided that includes step-by-step instructions for obtaining, installing, and using this VHDL simulator.

Modern digital systems are quite large. They contain complex logic circuits that would be difficult to design without using good CAD tools. Our treatment of VHDL should enable the reader to develop VHDL code that specifies logic circuits of varying degrees of complexity. To gain a proper appreciation of the design process, in addition to simulating the behavior of VHDL code (as mentioned above) it is highly beneficial to implement the designs in commercially-available PLDs by using CAD tools.

Some excellent CAD tools are available free of charge, for example, Intel Corporation's Quartus CAD software, which is widely used for implementing designs in programmable logic devices such as FPGAs. The Lite Edition of the Quartus software can be downloaded from Intel's website and used free-of-charge. Intel also provides a set of FPGA-based laboratory boards with accompanying tutorials and laboratory exercises that are appropriate for use with this book in a university setting. This material can be found on the Internet by searching for "Intel FPGA Academic Program".

SOLVED PROBLEMS

The chapters include examples of solved problems. They provide typical homework problems and show how they may be solved.

HOMEWORK PROBLEMS

More than 400 homework problems are provided in this book. Answers to selected problems are given at the back. Full solutions to all problems are available to instructors in the accompanying *Solutions Manual*.

POWERPOINT SLIDES AND SOLUTIONS MANUAL

A set of PowerPoint slides that contain all of the figures in the book is available on the Authors' website. Instructors can request access to these slides, as well as access to the *Solutions Manual* for the book, at:

www.mhhe.com/brownvranesic

ACKNOWLEDGMENTS

We wish to express our thanks to the people who have helped during the preparation of this book. Dan Vranesic produced a substantial amount of artwork. He and Deshanand Singh also helped with the preparation of the solutions manual, and Tom Czajkowski helped in checking the answers to problems. Matthew Brown edited the text to improve readability, and Andrew Brown helped to create some examples of VHDL code. The reviewers, William Barnes, New Jersey Institute of Technology; Thomas Bradicich, North Carolina State University; James Clark, McGill University; Stephen DeWeerth, Georgia Institute of Technology; Sander Eller, Cal Poly Pomona; Clay Gloster, Jr., North Carolina State University (Raleigh); Carl Hamacher, Queen's University; Vincent Heuring, University of Colorado; Yu Hen Hu, University of Wisconsin; Wei-Ming Lin, University of Texas (San Antonio); Wayne Loucks, University of Waterloo; Kartik Mohanram, Rice University; Jane Morehead, Mississippi State University; Chris Myers, University of Utah; Vojin Oklobdzija, University of California (Davis); James Palmer, Rochester Institute of Technology; Gandhi Puvvada, University of Southern California; Teodoro Robles, Milwaukee School of Engineering; Tatyana Roziner, Boston University; Rob Rutenbar, Carnegie Mellon University; Eric Schwartz, University of Florida; Wen-Tsong Shiue, Oregon State University; Peter Simko, Miami University; Scott Smith, University of Missouri (Rolla); Arun Somani, Iowa State University; Bernard Svihel, University of Texas (Arlington); and Zeljko Zilic, McGill University provided constructive criticism and made numerous suggestions for improvements.

Nicholas Reeder, Sinclair Community College, Dayton, OH, provided excellent proof reading.

Stephen Brown and Zvonko Vranesic

CONTENTS

Chapter 1
INTRODUCTION 1

1.1 Digital Hardware 2
- 1.1.1 Standard Chips 4
- 1.1.2 Programmable Logic Devices 4
- 1.1.3 Custom-Designed Chips 5

1.2 The Design Process 5
1.3 Structure of a Computer 7
1.4 Logic Circuit Design in This Book 10
1.5 Digital Representation of Information 10
- 1.5.1 Binary Numbers 11
- 1.5.2 Conversion between Decimal and Binary Systems 13
- 1.5.3 ASCII Character Code 14
- 1.5.4 Digital and Analog Information 14

1.6 Theory and Practice 16
Problems 17

Chapter 2
INTRODUCTION TO LOGIC CIRCUITS 19

2.1 Variables and Functions 20
2.2 Inversion 23
2.3 Truth Tables 24
2.4 Logic Gates and Networks 25
- 2.4.1 Analysis of a Logic Network 26

2.5 Boolean Algebra 30

2.5.1 The Venn Diagram 34
2.5.2 Notation and Terminology 38
2.5.3 Precedence of Operations 40
2.6 Synthesis Using AND, OR, and NOT Gates 40
2.6.1 Sum-of-Products and Product-of-Sums Forms 44
2.7 NAND and NOR Logic Networks 50
2.8 Design Examples 54
2.8.1 Three-Way Light Control 55
2.8.2 Multiplexer Circuit 56
2.8.3 Number Display 58
2.9 Introduction to CAD Tools 59
2.9.1 Design Entry 60
2.9.2 Logic Synthesis 61
2.9.3 Functional Simulation 62
2.9.4 Physical Design 62
2.9.5 Timing Analysis 62
2.9.6 Circuit Implementation 63
2.9.7 Complete Design Flow 63
2.10 Introduction to VHDL 63
2.10.1 Structural Specification of Logic Circuits 65
2.10.2 Behavioral Specification of Logic Circuits 69
2.10.3 Hierarchical VHDL Code 70
2.10.4 How not to Write VHDL Code 73
2.11 Minimization and Karnaugh Maps 73
2.12 Strategy for Minimization 81
2.12.1 Terminology 81
2.12.2 Minimization Procedure 83
2.13 Minimization of Product-of-Sums Forms 86
2.14 Incompletely Specified Functions 88
2.15 Multiple-Output Circuits 90
2.16 Concluding Remarks 95
2.17 Examples of Solved Problems 95
Problems 105
References 113

Chapter 3

NUMBER REPRESENTATION AND ARITHMETIC CIRCUITS 115

3.1 Positional Number Representation 116
- 3.1.1 Unsigned Integers 116
- 3.1.2 Octal and Hexadecimal Representations 116

3.2 Addition of Unsigned Numbers 118
- 3.2.1 Decomposed Full-Adder 122
- 3.2.2 Ripple-Carry Adder 122
- 3.2.3 Design Example 124

3.3 Signed Numbers 124
- 3.3.1 Negative Numbers 124
- 3.3.2 Addition and Subtraction 128
- 3.3.3 Adder and Subtractor Unit 131
- 3.3.4 Radix-Complement Schemes. 132
- 3.3.5 Arithmetic Overflow 136
- 3.3.6 Performance Issues 137

3.4 Fast Adders 138
- 3.4.1 Carry-Lookahead Adder 138

3.5 Design of Arithmetic Circuits Using CAD Tools 144
- 3.5.1 Design of Arithmetic Circuits Using Schematic Capture 144
- 3.5.2 Design of Arithmetic Circuits Using VHDL 144
- 3.5.3 Representation of Numbers in VHDL Code 147
- 3.5.4 Arithmetic Assignment Statements 149

3.6 Multiplication 152
- 3.6.1 Array Multiplier for Unsigned Numbers 154
- 3.6.2 Multiplication of Signed Numbers 154

3.7 Other Number Representations 156
- 3.7.1 Fixed-Point Numbers 157
- 3.7.2 Floating-Point Numbers 157
- 3.7.3 Binary-Coded-Decimal Representation 159

3.8 Examples of Solved Problems 163

Problems 169

Reference 172

Chapter 4

COMBINATIONAL-CIRCUIT BUILDING BLOCKS 173

4.1 Multiplexers 174
- 4.1.1 Synthesis of Logic Functions Using Multiplexers 177
- 4.1.2 Multiplexer Synthesis Using Shannon's Expansion 180

4.2 Decoders 184
- 4.2.1 Demultiplexers 188

4.3 Encoders 188
- 4.3.1 Binary Encoders 189
- 4.3.2 Priority Encoders 190

4.4 Code Converters 191

4.5 Arithmetic Comparison Circuits 192

4.6 VHDL for Combinational Circuits 193
- 4.6.1 Assignment Statements 193
- 4.6.2 Selected Signal Assignment 194
- 4.6.3 Conditional Signal Assignment 197
- 4.6.4 Generate Statements 200
- 4.6.5 Concurrent and Sequential Assignment Statements 202
- 4.6.6 Process Statement 203
- 4.6.7 Case Statement 208
- 4.6.8 VHDL Operators 210

4.7 Concluding Remarks 215

4.8 Examples of Solved Problems 215

Problems 226

Chapter 5
FLIP-FLOPS, REGISTERS, AND COUNTERS 231

5.1 Basic Latch 232

5.2 Gated SR Latch 234
- 5.2.1 Gated SR Latch with NAND Gates 236

5.3 Gated D Latch 237
- 5.3.1 Effects of Propagation Delays 239

5.4 Edge-Triggered D Flip-Flops 239
- 5.4.1 M-S D Flip-Flop 240
- 5.4.2 Other Types of Edge-Triggered D Flip-Flops 241

5.4.3 D Flip-Flops with Clear and Preset 243
5.4.4 Flip-Flop Timing Parameters 246
5.5 T Flip-Flop 247
5.6 JK Flip-Flop 248
5.7 Summary of Terminology 249
5.8 Registers 249
5.8.1 Shift Register 250
5.8.2 Parallel-Access Shift Register 251
5.9 Counters 252
5.9.1 Asynchronous Counters 252
5.9.2 Synchronous Counters 254
5.9.3 Counters with Parallel Load 258
5.10 Reset Synchronization 260
5.11 Other Types of Counters 261
5.11.1 BCD Counter 261
5.11.2 Ring Counter 262
5.11.3 Johnson Counter 265
5.11.4 Remarks on Counter Design 265
5.12 Using Storage Elements with CAD Tools 265
5.12.1 Including Storage Elements in Schematics 265
5.12.2 Using VHDL Constructs for Storage Elements 268
5.13 Using VHDL Sequential Statements for Registers and Counters 273
5.13.1 Flip-Flops and Registers with Enable Inputs 280
5.13.2 Shift Registers with Enable Inputs 283
5.14 Design Example - Reaction Timer 283
5.14.1 Register Transfer Level (RTL) Code 290
5.15 Timing Analysis of Flip-flop Circuits 292
5.15.1 Timing Analysis with Clock Skew 294
5.16 Concluding Remarks 296
5.17 Examples of Solved Problems 296
Problems 303

Chapter 6
SYNCHRONOUS SEQUENTIAL CIRCUITS 309

6.1 Basic Design Steps 311

6.1.1 State Diagram 311
6.1.2 State Table 313
6.1.3 State Assignment 313
6.1.4 Choice of Flip-Flops and Derivation of Next-State and Output Expressions 315
6.1.5 Timing Diagram 316
6.1.6 Summary of Design Steps 318
6.2 State-Assignment Problem 322
6.2.1 One-Hot Encoding 325
6.3 Mealy State Model 327
6.4 Design of Finite State Machines Using CAD Tools 331
6.4.1 VHDL Code for Moore-Type FSMs 332
6.4.2 Synthesis of VHDL Code 334
6.4.3 Simulating and Testing the Circuit 335
6.4.4 An Alternative Style of VHDL Code 337
6.4.5 Summary of Design Steps When Using CAD Tools 338
6.4.6 Specifying the State Assignment in VHDL Code 338
6.4.7 Specification of Mealy FSMs Using VHDL 341
6.5 Serial Adder Example 343
6.5.1 Mealy-Type FSM for Serial Adder 343
6.5.2 Moore-Type FSM for Serial Adder 345
6.5.3 VHDL Code for the Serial Adder 347
6.6 State Minimization 351
6.6.1 Partitioning Minimization Procedure 353
6.6.2 Incompletely Specified FSMs 360
6.7 Design of a Counter Using the Sequential Circuit Approach 361
6.7.1 State Diagram and State Table for a Modulo-8 Counter 362
6.7.2 State Assignment 362
6.7.3 Implementation Using D-Type Flip-Flops 363
6.7.4 Implementation Using JK-Type Flip-Flops 365
6.7.5 Example—A Different Counter 368
6.8 FSM as an Arbiter Circuit 371
6.9 Analysis of Synchronous Sequential Circuits 376
6.10 Algorithmic State Machine (ASM) Charts 380
6.11 Formal Model for Sequential Circuits 383
6.12 Concluding Remarks 385
6.13 Examples of Solved Problems 385

Problems 394

Chapter 7
DIGITAL SYSTEM DESIGN 399

7.1 Bus Structure 400
7.1.1 Using Tri-State Drivers to Implement a Bus 400
7.1.2 Using Multiplexers to Implement a Bus 402
7.1.3 VHDL Code for Specification of Bus Structures 404
7.2 Simple Processor 408
7.3 A Bit-Counting Circuit 420
7.4 Shift-and-Add Multiplier 425
7.5 Divider 432
7.6 Arithmetic Mean 442
7.7 Sort Operation 445
7.8 Clock Synchronization and Timing Issues 454
7.8.1 Clock Distribution 456
7.8.2 Flip-Flop Timing Parameters 457
7.8.3 Asynchronous Inputs to Flip-Flops 458
7.8.4 Switch Debouncing 459
7.9 Concluding Remarks 461
Problems 461

Chapter 8
OPTIMIZED IMPLEMENTATION OF LOGIC FUNCTIONS 465

8.1 Multilevel Synthesis 466
8.1.1 Factoring 467
8.1.2 Functional Decomposition 470
8.1.3 Multilevel NAND and NOR Circuits 475
8.2 Analysis of Multilevel Circuits 478
8.3 Alternative Representations of Logic Functions 483
8.3.1 Cubical Representation 483
8.3.2 Binary Decision Diagrams 487

8.4 Optimization Techniques Based on Cubical Representation 493
8.4.1 A Tabular Method for Minimization 493
8.4.2 A Cubical Technique for Minimization 501
8.4.3 Practical Considerations 508
8.5 Concluding Remarks 509
8.6 Examples of Solved Problems 509
Problems 518

Chapter 9
ASYNCHRONOUS SEQUENTIAL CIRCUITS 521

9.1 Asynchronous Behavior 522
9.2 Analysis of Asynchronous Circuits 525
9.3 Synthesis of Asynchronous Circuits 533
9.4 State Reduction 546
9.5 State Assignment 560
9.5.1 Transition Diagram 563
9.5.2 Exploiting Unspecified Next-State Entries 566
9.5.3 State Assignment Using Additional State Variables 569
9.5.4 One-Hot State Assignment 575
9.6 Hazards 576
9.6.1 Static Hazards 576
9.6.2 Dynamic Hazards 581
9.6.3 Significance of Hazards 583
9.7 A Complete Design Example 583
9.7.1 The Vending-Machine Controller 583
9.8 Concluding Remarks 588
9.9 Examples of Solved Problems 590
Problems 598

Chapter 10
COMPUTER AIDED DESIGN TOOLS 603

10.1 Synthesis 604
10.1.1 Netlist Generation 604
10.1.2 Gate Optimization 604

10.1.3 Technology Mapping 606
10.2 Physical Design 609
10.2.1 Placement 609
10.2.2 Routing 611
10.2.3 Timing Analysis 613
10.3 Concluding Remarks 616
References 616

Chapter 11
TESTING OF LOGIC CIRCUITS 617

11.1 Fault Model 618
11.1.1 Stuck-at Model 618
11.1.2 Single and Multiple Faults 619
11.1.3 CMOS Circuits 619
11.2 Complexity of a Test Set 619
11.3 Path Sensitizing 621
11.3.1 Detection of a Specific Fault 623
11.4 Circuits with Tree Structure 625
11.5 Random Tests 626
11.6 Testing of Sequential Circuits 629
11.6.1 Design for Testability 629
11.7 Built-in Self-Test 633
11.7.1 Built-in Logic Block Observer 637
11.7.2 Signature Analysis 639
11.7.3 Boundary Scan 640
11.8 Printed Circuit Boards 640
11.8.1 Testing of PCBs 642
11.8.2 Instrumentation 643
11.9 Concluding Remarks 644
Problems 644
References 647

Appendix A
VHDL REFERENCE 649

A.1 Documentation in VHDL Code 650
A.2 Data Objects 650
A.2.1 Data Object Names 650
A.2.2 Data Object Values and Numbers 650
A.2.3 SIGNAL Data Objects 651
A.2.4 BIT and BIT VECTOR Types 651
A.2.5 STD LOGIC and STD LOGIC VECTOR Types 652
A.2.6 STD ULOGIC Type 652
A.2.7 SIGNED and UNSIGNED Types 653
A.2.8 INTEGER Type 653
A.2.9 BOOLEAN Type 654
A.2.10 ENUMERATION Type 654
A.2.11 CONSTANT Data Objects 654
A.2.12 VARIABLE Data Objects 655
A.2.13 Type Conversion 655
A.2.14 Arrays 655
A.3 Operators 656
A.4 VHDL Design Entity 657
A.4.1 ENTITY Declaration 657
A.4.2 Architecture 658
A.5 Package 660
A.6 Using Subcircuits 661
A.6.1 Declaring a COMPONENT in a Package 663
A.7 Concurrent Assignment Statements 664
A.7.1 Simple Signal Assignment 664
A.7.2 Assigning Signal Values Using OTHERS 665
A.7.3 Selected Signal Assignment 666
A.7.4 Conditional Signal Assignment 667
A.7.5 GENERATE Statement 668
A.8 Defining an Entity with GENERICs 668
A.9 Sequential Assignment Statements 669
A.9.1 PROCESS Statement 671
A.9.2 IF Statement 671
A.9.3 CASE Statement 672
A.9.4 Loop Statements 673
A.9.5 Using a Process for a Combinational Circuit 673
A.9.6 Statement Ordering 674

A.9.7 Using a VARIABLE in a PROCESS 675
A.10 Sequential Circuits 680
A.10.1 A Gated D Latch 680
A.10.2 D Flip-Flop 681
A.10.3 Using a WAIT UNTIL Statement 681
A.10.4 A Flip-Flop with Asynchronous Reset 682
A.10.5 Synchronous Reset 682
A.10.6 Registers 684
A.10.7 Shift Registers 686
A.10.8 Counters 688
A.10.9 Using Subcircuits with GENERIC Parameters 688
A.10.10 A Moore-Type Finite State Machine 692
A.10.11 A Mealy-Type Finite State Machine 696
A.11 Common Errors in VHDL Code 696
A.12 Concluding Remarks 699
References 700

Appendix B
IMPLEMENTATION TECHNOLOGY 701

B.1 Transistor Switches 702
B.2 NMOS Logic Gates 704
B.3 CMOS Logic Gates 707
B.3.1 Speed of Logic Gate Circuits 714
B.4 Negative Logic System 715
B.5 Standard Chips 718
B.5.1 7400-Series Standard Chips 718
B.6 Programmable Logic Devices 720
B.6.1 Programmable Logic Array (PLA) 721
B.6.2 Programmable Array Logic (PAL) 723
B.6.3 Programming of PLAs and PALs 725
B.6.4 Complex Programmable Logic Devices (CPLDs) 727
B.6.5 Field-Programmable Gate Arrays 731
B.7 Custom Chips, Standard Cells, and Gate Arrays 736
B.8 Practical Aspects 738
B.8.1 MOSFET Fabrication and Behavior 738

B.8.2 MOSFET On-Resistance 742
B.8.3 Voltage Levels in Logic Gates 743
B.8.4 Noise Margin 745
B.8.5 Dynamic Operation of Logic Gates 746
B.8.6 Power Dissipation in Logic Gates 749
B.8.7 Passing 1s and 0s Through Transistor Switches 751
B.8.8 Transmission Gates 753
B.8.9 Fan-in and Fan-out in Logic Gates 755
B.8.10 Tri-state Drivers 759
B.9 Static Random Access Memory (SRAM) 761
B.9.1 SRAM Blocks in PLDs 762
B.10 Implementation Details for SPLDs, CPLDs, and FPGAs 763
B.10.1 Implementation in FPGAs 771
B.11 Concluding Remarks 773
B.12 Examples of Solved Problems 773
Problems 780
References 790

ANSWERS 791

INDEX 805

Mc Graw Hill
501-1111

Instructors: Student Success Starts with You

Tools to enhance your unique voice

Want to build your own course? No problem. Prefer to use an OLC-aligned, prebuilt course? Easy. Want to make changes throughout the semester? Sure. And you'll save time with Connect's auto-grading too.

65%
Less Time Grading

Laptop: McGraw Hill; Woman/dog: George Doyle/Getty Images

Study made personal

Incorporate adaptive study resources like SmartBook® 2.0 into your course and help your students be better prepared in less time. Learn more about the powerful personalized learning experience available in SmartBook 2.0 at **www.mheducation.com/highered/connect/smartbook**

Affordable solutions, added value

Make technology work for you with LMS integration for single sign-on access, mobile access to the digital textbook, and reports to quickly show you how each of your students is doing. And with our Inclusive Access program you can provide all these tools at a discount to your students. Ask your McGraw Hill representative for more information.

Padlock: Jobalou/Getty Images

Solutions for your challenges

A product isn't a solution. Real solutions are affordable, reliable, and come with training and ongoing support when you need it and how you want it. Visit **www.supportateverystep.com** for videos and resources both you and your students can use throughout the semester.

Checkmark: Jobalou/Getty Images

SUPPORT AT every step

Students: Get Learning that Fits You

Effective tools for efficient studying

Connect is designed to help you be more productive with simple, flexible, intuitive tools that maximize your study time and meet your individual learning needs. Get learning that works for you with Connect.

Study anytime, anywhere

Download the free ReadAnywhere app and access your online eBook, SmartBook 2.0, or Adaptive Learning Assignments when it's convenient, even if you're offline. And since the app automatically syncs with your Connect account, all of your work is available every time you open it. Find out more at **www.mheducation.com/readanywhere**

"I really liked this app—it made it easy to study when you don't have your text-book in front of you."

- Jordan Cunningham,
Eastern Washington University

Calendar: owattaphotos/Getty Images

Everything you need in one place

Your Connect course has everything you need—whether reading on your digital eBook or completing assignments for class, Connect makes it easy to get your work done.

Learning for everyone

McGraw Hill works directly with Accessibility Services Departments and faculty to meet the learning needs of all students. Please contact your Accessibility Services Office and ask them to email accessibility@mheducation.com, or visit **www.mheducation.com/about/accessibility** for more information.

Top: Jenner Images/Getty Images, Left: Hero Images/Getty Images, Right: Hero Images/Getty Images

chapter

1

INTRODUCTION

CHAPTER OBJECTIVES

In this chapter you will be introduced to:

- Digital hardware components
- An overview of the design process
- Binary numbers
- Digital representation of information

This book is about logic circuits—the circuits from which computers are built. Proper understanding of logic circuits is vital for today's electrical and computer engineers. These circuits are the key ingredient of computers and are also used in many other applications. They are found in commonly used products like music and video players, electronic games, digital watches, cameras, televisions, printers, and many household appliances, as well as in large systems, such as telephone networks, Internet equipment, television broadcast equipment, industrial control units, and medical instruments. In short, logic circuits are an important part of almost all modern products.

The material in this book will introduce the reader to the many issues involved in the design of logic circuits. It explains the key ideas with simple examples and shows how complex circuits can be derived from elementary ones. We cover the classical theory used in the design of logic circuits because it provides the reader with an intuitive understanding of the nature of such circuits. But, throughout the book, we also illustrate the modern way of designing logic circuits using sophisticated *computer aided design (CAD)* software tools. The CAD methodology adopted in the book is based on the industry-standard design language called *VHDL*. Design with VHDL is first introduced in Chapter 2, and usage of VHDL and CAD tools is an integral part of each chapter in the book.

Logic circuits are implemented electronically, using transistors on an integrated circuit chip. Commonly available chips that use modern technology may contain more than a billion transistors, as in the case of some computer processors. The basic building blocks for such circuits are easy to understand, but there is nothing simple about a circuit that contains billions of transistors. The complexity that comes with large circuits can be handled successfully only by using highly organized design techniques. We introduce these techniques in this chapter, but first we briefly describe the hardware technology used to build logic circuits.

1.1 Digital Hardware

Logic circuits are used to build computer hardware, as well as many other types of products. All such products are broadly classified as *digital hardware*. The reason that the name *digital* is used will be explained in Section 1.5—it derives from the way in which information is represented in computers, as electronic signals that correspond to digits of information.

The technology used to build digital hardware has evolved dramatically over the past few decades. Until the 1960s logic circuits were constructed with bulky components, such as transistors and resistors that came as individual parts. The advent of integrated circuits made it possible to place a number of transistors, and thus an entire circuit, on a single chip. In the beginning these circuits had only a few transistors, but as the technology improved they became more complex. Integrated circuit chips are

manufactured on a silicon wafer, such as the one shown in Figure 1.1. The wafer is cut to produce the individual chips, which are then placed inside a special type of chip package. By 1970 it was possible to implement all circuitry needed to realize a microprocessor on a single chip. Although early microprocessors had modest computing capability by today's standards, they opened the door for the information processing revolution by providing the means for implementation of affordable personal computers.

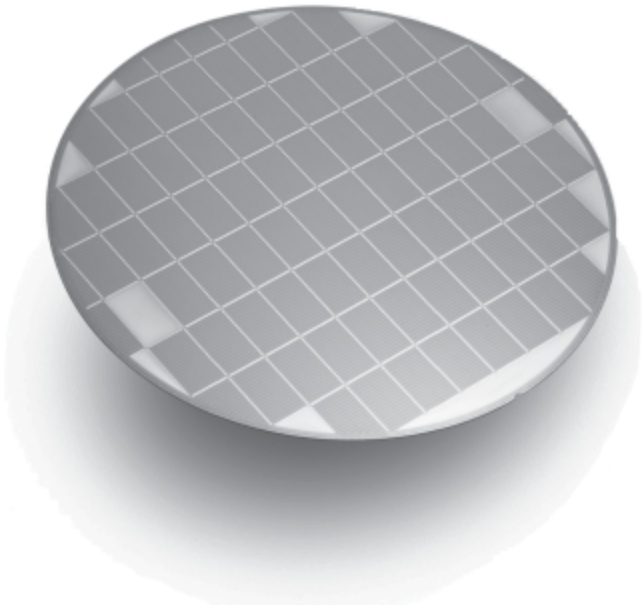


Figure 1.1 A silicon wafer.

About 30 years ago Gordon Moore, chairman of Intel Corporation, observed that integrated circuit technology was progressing at an astounding rate, approximately doubling the number of transistors that could be placed on a chip every two years. This phenomenon is informally known as *Moore's law*. Thus in the early 1990s microprocessors could be manufactured with a few million transistors, and by the late 1990s it became possible to fabricate chips that had tens of millions of transistors. Presently, chips can be manufactured containing billions of transistors.

Moore's law is expected to continue to hold true for a number of years. A consortium of integrated circuit associations produces a forecast of how the technology is expected to evolve. Known as the *International Roadmap for Devices and Systems (IRDS)*, this forecast discusses many aspects of technology, including the maximum number of transistors that can be

manufactured in a given chip area. An example of data that can be derived from the roadmap is given in Figure 1.2. It indicates how the maximum number of transistors that can be manufactured on a single chip has changed over time—chips with about 10 million transistors could be successfully manufactured in 1995, and this number has steadily increased, leading to modern chips that have over a billion transistors. The roadmap predicts that chips with more than 100 billion transistors will be possible by the year 2025. There is no doubt that this technology will have a huge impact on all aspects of people's lives.

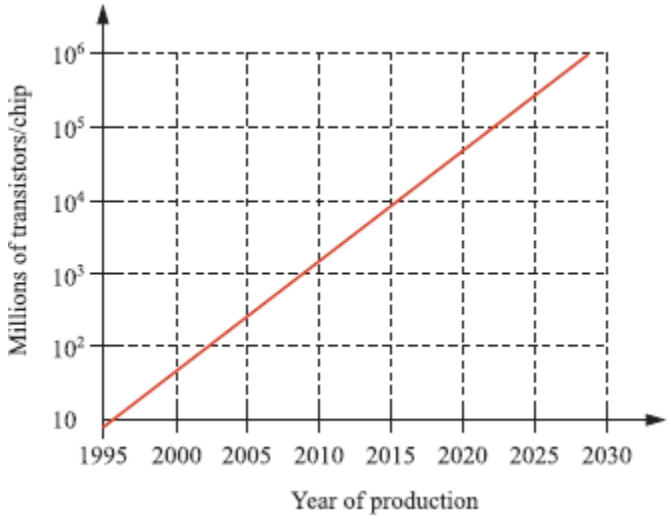


Figure 1.2 An estimate of the maximum number of transistors per chip over time.

The designer of digital hardware may be faced with designing logic circuits that can be implemented on a single chip or designing circuits that involve a number of chips placed on a *printed circuit board (PCB)*. Frequently, some of the logic circuits can be realized in existing chips that are readily available. This situation simplifies the design task and shortens the time needed to develop the final product. Before we discuss the design process in detail, we should introduce the different types of integrated circuit chips that may be used.

There exists a large variety of chips that implement various functions that are useful in the design of digital hardware. The chips range from simple ones with low functionality to extremely complex chips. For example, a digital hardware product may require a microprocessor to perform some arithmetic operations, memory chips to provide storage capability, and

interface chips that allow easy connection to input and output devices. Such chips are available from various vendors.

For many digital hardware products, it is also necessary to design and build some logic circuits from scratch. For implementing these circuits, three main types of chips may be used: standard chips, programmable logic devices, and custom chips. These are discussed next.

1.1.1 STANDARD CHIPS

Numerous chips are available that realize some commonly used logic circuits. We will refer to these as *standard chips*, because they usually conform to an agreed-upon standard in terms of functionality and physical configuration. Each standard chip contains a small amount of circuitry (usually involving fewer than 100 transistors) and performs a simple function. To build a logic circuit, the designer chooses the chips that perform whatever functions are needed and then defines how these chips should be interconnected to realize a larger logic circuit.

Standard chips were popular for building logic circuits until the early 1980s. However, as integrated circuit technology improved, it became inefficient to use valuable space on PCBs for chips with low functionality. Another drawback of standard chips is that the functionality of each chip is fixed and cannot be changed.

1.1.2 PROGRAMMABLE LOGIC DEVICES

In contrast to standard chips that have fixed functionality, it is possible to construct chips that contain circuitry which can be configured by the user to implement a wide range of different logic circuits. These chips have a very general structure and include a collection of *programmable switches* that allow the internal circuitry in the chip to be configured in many different ways. The designer can implement whatever functions are required for a particular application by setting the programmable switches as needed. The switches are programmed by the end user, rather than when the chip is manufactured. Such chips are known as *programmable logic devices (PLDs)*.

PLDs are available in a wide range of sizes, and can be used to implement very large logic circuits. The most commonly used type of PLD is known as a *field-programmable gate array (FPGA)*. The largest FPGAs

contain billions of transistors and support the implementation of complex digital systems. An FPGA consists of a large number of small logic circuit elements, which can be connected together by using programmable switches in the FPGA. Because of their high capacity, and their capability to be tailored to meet the requirements of a specific application, FPGAs are widely used today.

1.1.3 CUSTOM-DESIGNED CHIPS

FPGAs are available as off-the-shelf components that can be purchased from different suppliers. Because they are programmable, they can be used to implement most logic circuits found in digital hardware. However, they also have a drawback in that the programmable switches consume valuable chip area and limit the speed of operation of implemented circuits. Thus in some cases FPGAs may not meet the desired performance or cost objectives. In such situations it is possible to design a chip from scratch; namely, the logic circuitry that must be included on the chip is designed first and then the chip is manufactured by a company that has the fabrication facilities. This approach is known as *custom* or *semi-custom design*, and such chips are often called *application-specific integrated circuits (ASICs)*.

The main advantage of a custom chip is that its design can be optimized for a specific task; hence it usually leads to better performance. It is possible to include a larger amount of logic circuitry in a custom chip than would be possible in other types of chips. The cost of producing such chips is high, but if they are used in a product that is sold in large quantities, then the cost per chip, amortized over the total number of chips fabricated, may be lower than the total cost of off-the-shelf chips that would be needed to implement the same function(s). Moreover, if a single chip can be used instead of multiple chips to achieve the same goal, then a smaller area is needed on a PCB that houses the chips in the final product. This results in a further reduction in cost.

A disadvantage of the custom-design approach is that manufacturing a custom chip often takes a considerable amount of time, on the order of months. In contrast, if an FPGA can be used instead, then the chips are programmed by the end user and no manufacturing delays are involved.

1.2 The Design Process

The availability of computer-based tools has greatly influenced the design process in a wide variety of environments. For example, designing an automobile is similar in the general approach to designing a furnace or a computer. Certain steps in the development cycle must be performed if the final product is to meet the specified objectives.

The flowchart in Figure 1.3 depicts a typical development process. We assume that the process is to develop a product that meets certain expectations. The most obvious requirements are that the product must function properly, that it must meet an expected level of performance, and that its cost should not exceed a given target.

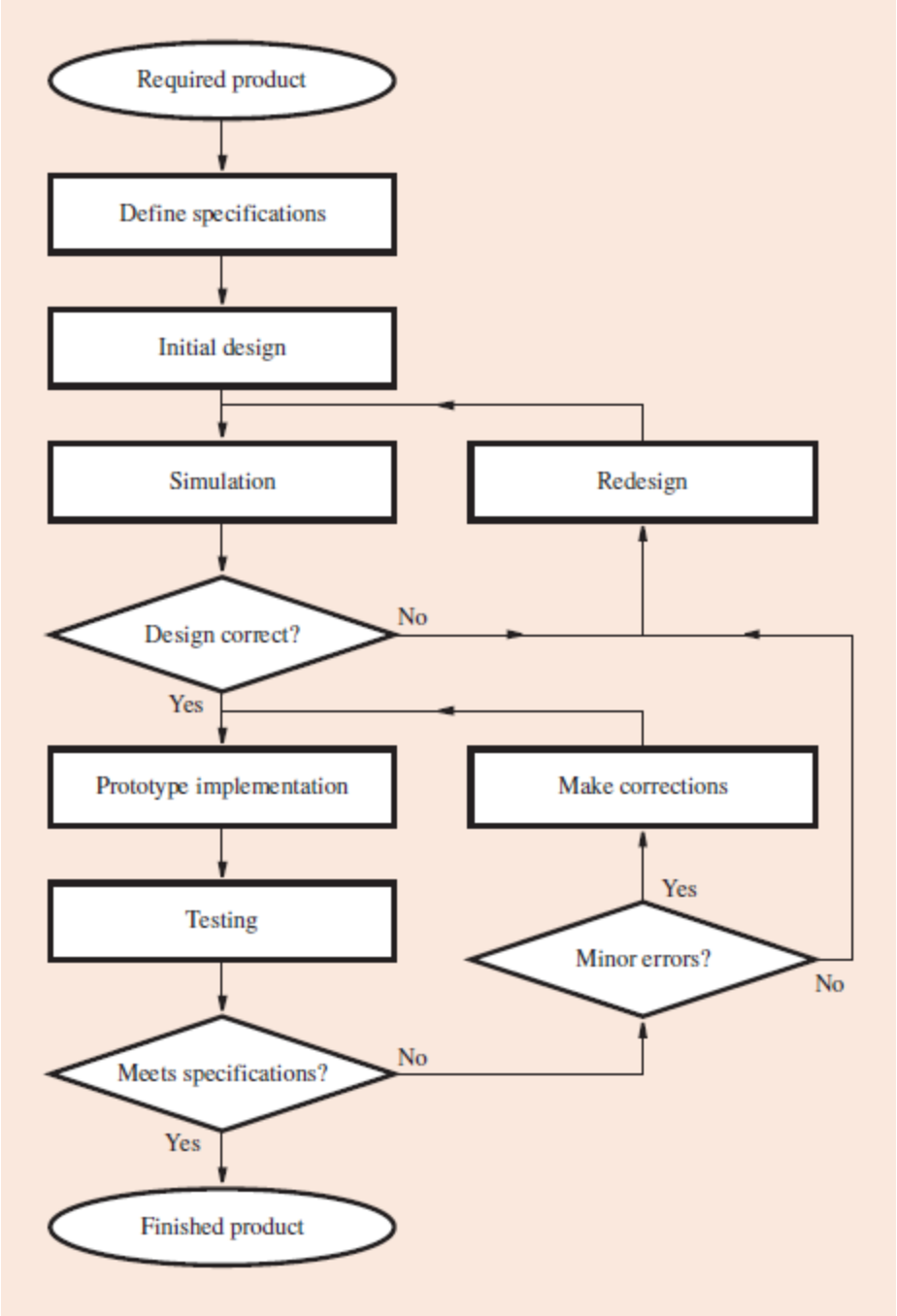


Figure 1.3 The development process.

The process begins with the definition of product specifications. The essential features of the product are identified, and an acceptable method of evaluating the implemented features in the final product is established. The specifications must be tight enough to ensure that the developed product will meet the general expectations, but should not be unnecessarily

constraining (that is, the specifications should not prevent design choices that may lead to unforeseen advantages).

From a complete set of specifications, it is necessary to define the general structure of an initial design of the product. This step is difficult to automate. It is usually performed by a human designer because there is no clear-cut strategy for developing a product's overall structure—it requires considerable design experience and intuition.

After the general structure is established, CAD tools are used to work out the details. Many types of CAD tools are available, ranging from those that help with the design of individual parts of the system to those that allow the entire system's structure to be represented in a computer. When the initial design is finished, the results must be verified against the original specifications. Traditionally, before the advent of CAD tools, this step involved constructing a physical model of the designed product, usually including just the key parts. Today it is seldom necessary to build a physical model. CAD tools enable designers to simulate the behavior of incredibly complex products, and such simulations are used to determine whether the obtained design meets the required specifications. If errors are found, then appropriate changes are made and the verification of the new design is repeated through simulation. Although some design flaws may escape detection via simulation, usually all but the most subtle problems are discovered in this way.

When the simulation indicates that the design is correct, a complete physical prototype of the product is constructed. The prototype is thoroughly tested for conformance with the specifications. Any errors revealed in the testing must be fixed. The errors may be minor, and often they can be eliminated by making small corrections directly on the prototype of the product. In case of large errors, it is necessary to redesign the product and repeat the steps explained above. When the prototype passes all the tests, then the product is deemed to be successfully designed and it can go into production.

1.3 Structure of a Computer

To understand the role that logic circuits play in digital systems, consider the structure of a typical computer, as illustrated in Figure 1.4*a*. The computer case houses a number of printed circuit boards (PCBs), a power supply, and (not shown in the figure) storage units, like a hard disk and DVD or CD-ROM drives. Each unit is plugged into a main PCB, called the *motherboard*. As indicated on the bottom of the figure, the motherboard holds several integrated circuit chips, and it provides slots for connecting other PCBs, such as audio, video, and network boards.

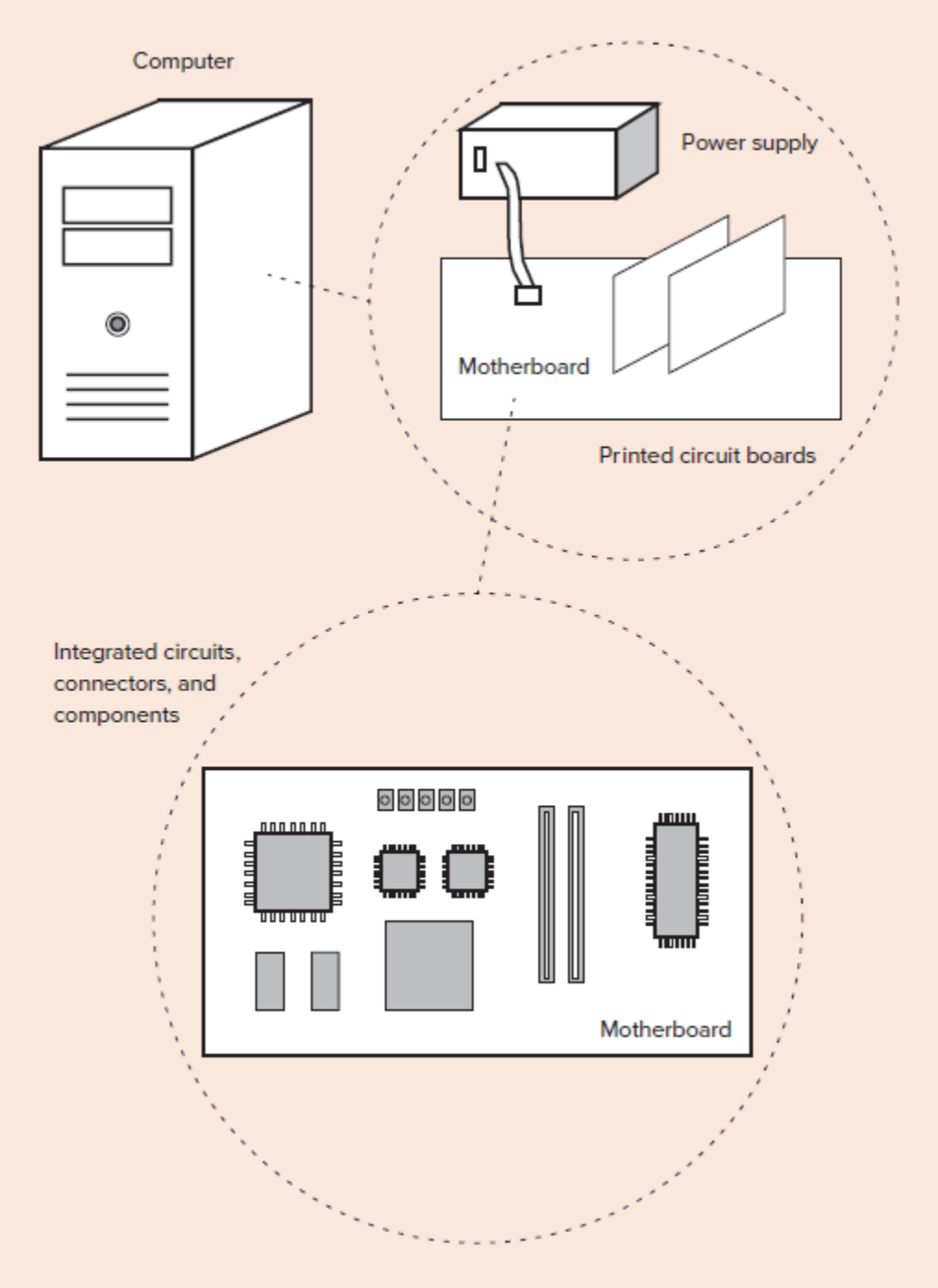


Figure 1.4 A digital hardware system (Part *a*)

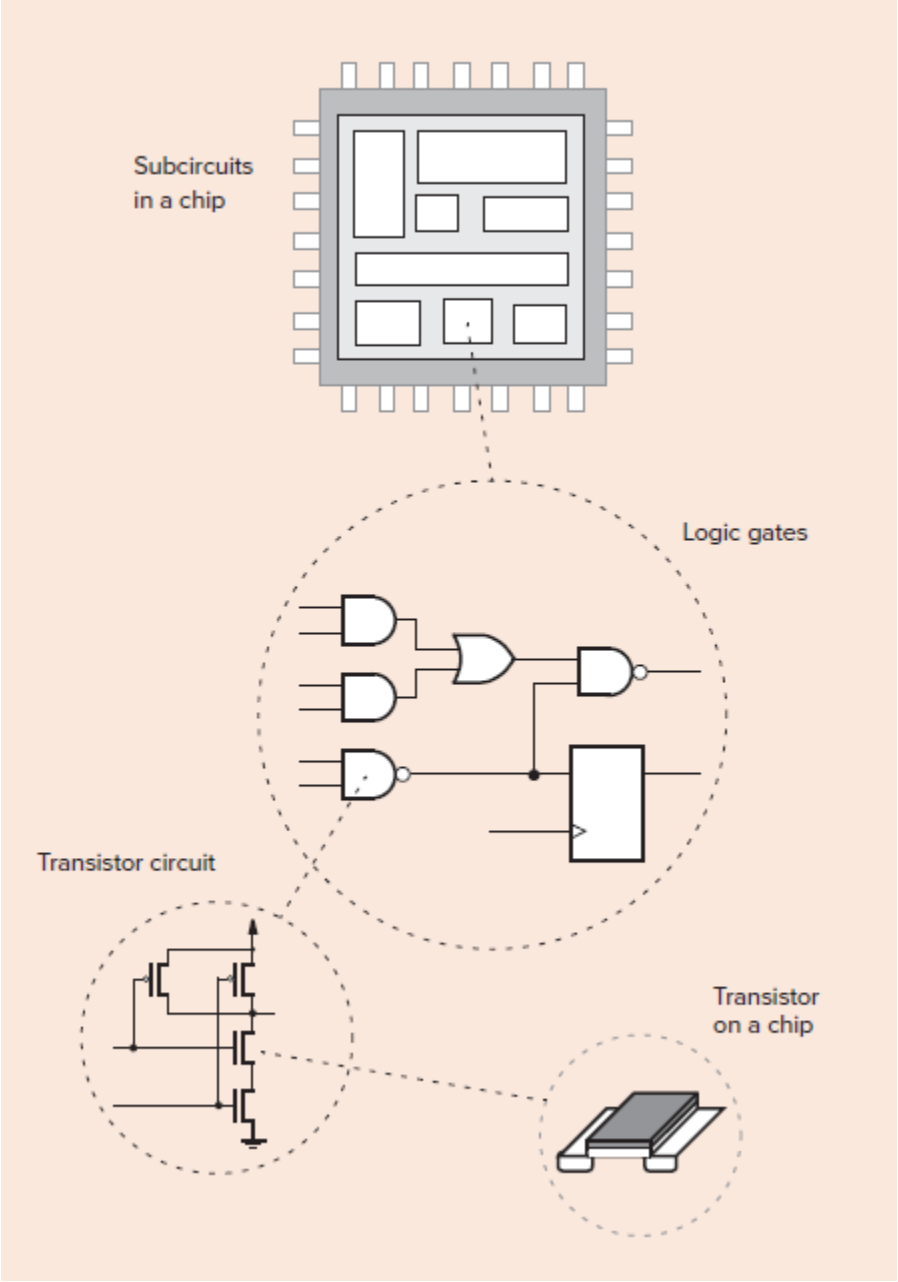


Figure 1.4 A digital hardware system (Part *b*)

Figure 1.4*b* illustrates the structure of an integrated circuit chip. The chip comprises a number of subcircuits, which are interconnected to build the complete circuit. Examples of subcircuits are those that perform arithmetic operations, store data, or control the flow of data. Each of these subcircuits is a logic circuit. As shown in the middle of the figure, a logic circuit comprises a network of connected *logic gates*. Each logic gate performs a very simple function, and more complex operations are realized by

connecting gates together. Logic gates are built with transistors, which in turn are implemented by fabricating various layers of material on a silicon chip.

This book is primarily concerned with the center portion of Figure 1.4*b*—the design of logic circuits. We explain how to design circuits that perform important functions, such as adding, subtracting, or multiplying numbers, counting, storing data, and controlling the processing of information. We show how the behavior of such circuits is specified, how the circuits are designed for minimum cost or maximum speed of operation, and how the circuits can be tested to ensure correct operation. We also briefly explain how transistors operate, and how they are built on silicon chips.

1.4 Logic Circuit Design in This Book

In this book we use a modern design approach based on the VHDL hardware description language and CAD tools to illustrate many aspects of logic circuit design. We selected this technology because it is widely used in industry and because it enables the readers to implement their designs in FPGA chips, as discussed below. This technology is particularly well-suited for educational purposes because many readers have access to facilities for using CAD tools and programming FPGA devices.

To gain practical experience and a deeper understanding of logic circuits, we advise the reader to implement the examples in this book using CAD software. Most of the major vendors of CAD systems provide their software at no cost to university students for educational use. Some examples are Intel, Cadence, Mentor Graphics, Synopsys, and Xilinx. The CAD systems offered by any of these companies can be used equally well with this book. Two examples of CAD systems that are particularly well-suited for use with this book are those provided by Intel and Xilinx. The CAD tools from both of these corporations support all phases of the design cycle for logic circuits and are powerful and easy to use. The reader is encouraged to visit the website for these companies, where the software tools and tutorials that explain their use can be downloaded and installed onto any personal computer.

To facilitate experimentation with logic circuits, some FPGA manufacturers provide special PCBs that include one or more FPGA chips and an interface to a personal computer. Once a logic circuit has been designed using the CAD tools, the circuit can be *programmed* into an FPGA on the board. Inputs can then be applied to the FPGA by way of switches and other devices, and the generated outputs can be examined. An example of such a board is depicted in Figure 1.5. This type of board is an excellent vehicle for learning about logic circuits, because it provides a collection of simple input and output devices. Many illustrative experiments can be carried out by designing and implementing logic circuits using the FPGA chip on the board.

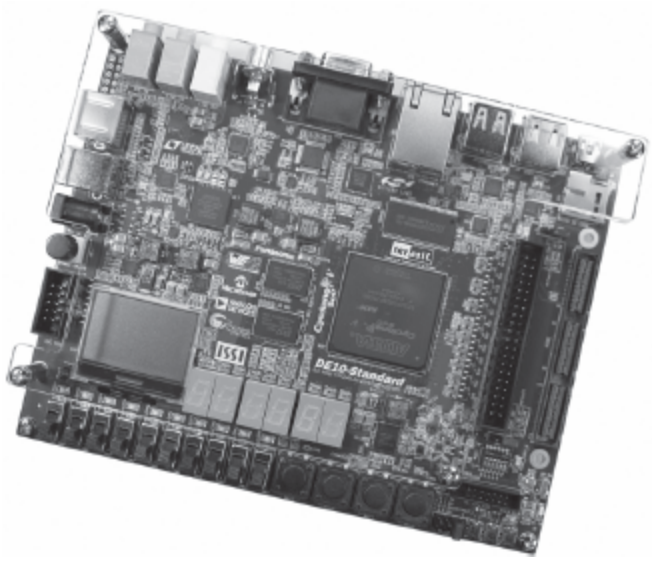


Figure 1.5 An FPGA board.

1.5 Digital Representation of Information

In Section 1.1 we mentioned that information is represented in logic circuits as electronic signals. Each of these signals can be thought of as representing one *digit* of information. To make the design of logic circuits easier, each digit is allowed to take on only two possible values, usually denoted as 0 and 1. These logic values are implemented as voltage levels in a circuit; the value 0 is usually represented as 0 V (ground), and the value 1 is the

voltage level of the circuit's power supply. As we discuss in Appendix B, typical power-supply voltages in logic circuits range from 1 V DC to 5 V DC.

In general, all information in logic circuits is represented as combinations of 0 and 1 digits. Before beginning our discussion of logic circuits in Chapter 2, it will be helpful to examine how numbers, alphanumeric data (text), and other information can be represented using the digits 0 and 1.

1.5.1 BINARY NUMBERS

In the familiar decimal system, a number consists of digits that have 10 possible values, from 0 to 9, and each digit represents a multiple of a power of 10. For example, the number 8547 represents $8 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$. We do not normally write the powers of 10 with the number, because they are implied by the positions of the digits. In general, a decimal integer is expressed by an *n*-tuple comprising *n* decimal digits

$$D = d_{n-1}d_{n-2} \cdots d_1 d_0$$

which represents the value

$$V(D) = d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \cdots + d_1 \times 10^1 + d_0 \times 10^0$$

This is referred to as the *positional number representation*.

Because the digits have 10 possible values and each digit is weighted as a power of 10, we say that decimal numbers are *base*-10 numbers. Decimal numbers are familiar, convenient, and easy to understand. However, since digital circuits represent information using only the values 0 and 1, it is not practical to have digits that can assume ten values. In these circuits it is more appropriate to use the binary, or *base*-2, system which has only the digits 0 and 1. Each binary digit is called a *bit*. In the binary number system, the same positional number representation is used so that

$$B = b_{n-1}b_{n-2} \cdots b_1 b_0$$

represents an integer that has the value

$$V(B) = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \quad [1.1]$$
$$= \sum_{i=0}^{n-1} b_i \times 2^i$$

For example, the binary number 1101 represents the value

$$V = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Because a particular digit pattern has different meanings for different bases, we will indicate the base as a subscript when there is potential for confusion. Thus to specify that 1101 is a base-2 number, we will write $(1101)_2$. Evaluating the preceding expression for V gives $V = 8 + 4 + 1 = 13$. Hence

$$(1101)_2 = (13)_{10}$$

The range of integers that can be represented by a binary number depends on the number of bits used. Table 1.1 lists the first 15 positive integers and shows their binary representations using four bits. An example of a larger number is $(10110111)_2 = (183)_{10}$. In general, using n bits allows representation of positive integers in the range 0 to $2^n - 1$.

Table 1-1 Numbers in decimal and binary.

Decimal representation	Binary representation
00	0000
01	0001
02	0010
03	0011
04	0100
05	0101
06	0110
07	0111
08	1000
09	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

In a binary number the rightmost bit is usually referred to as the *least-significant bit (LSB)*. The left-most bit, which has the highest power of 2 associated with it, is called the *most-significant bit (MSB)*. In digital systems it is often convenient to consider several bits together as a group. A group of four bits is called a *nibble*, and a group of eight bits is called a *byte*.

1.5.2 CONVERSION BETWEEN DECIMAL AND BINARY SYSTEMS

A binary number is converted into a decimal number simply by applying Equation 1.1 and evaluating it using decimal arithmetic. Converting a decimal number into a binary number is not quite as straightforward, because we need to construct the number by using powers of 2. For example, the number $(17)_{10}$ is $2^4 + 2^0 = (10001)_2$, and the number $(50)_{10}$ is $2^5 + 2^4 + 2^1 = (110010)_2$. In general, the conversion can be performed by successively dividing the decimal number by 2 as follows. Suppose that a decimal number $D = d_{k-1} \cdots d_1 d_0$, with a value V, is to be converted into a binary number $B = b_{n-1} \cdots b_2 b_1 b_0$. Then, we can write V in the form

$$V = b_{n-1} \times 2^{n-1} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0$$

If we now divide V by 2, the result is

$$\frac{V}{2} = b_{n-1} \times 2^{n-2} + \cdots + b_2 \times 2^1 + b_1 + \frac{b_0}{2}$$

The quotient of this integer division is $b_{n-1} \times 2^{n-2} + \cdots + b_2 \times 2 + b_1$, and the remainder is b_0. If the remainder is 0, then $b_0 = 0$; if it is 1, then $b_0 = 1$. Observe that the quotient is just another binary number, which comprises $n - 1$ bits, rather than n bits. Dividing this number by 2 yields the remainder b_1. The new quotient is

$$b_{n-1} \times 2^{n-3} + \cdots + b_2$$

Continuing the process of dividing the new quotient by 2, and determining one bit in each step, will produce all bits of the binary number. The process continues until the quotient becomes 0. Figure 1.6 illustrates the conversion process, using the example Convert $(857)_{10}$. Note that the least-significant bit

(LSB) is generated first and the most-significant bit (MSB) is generated last.

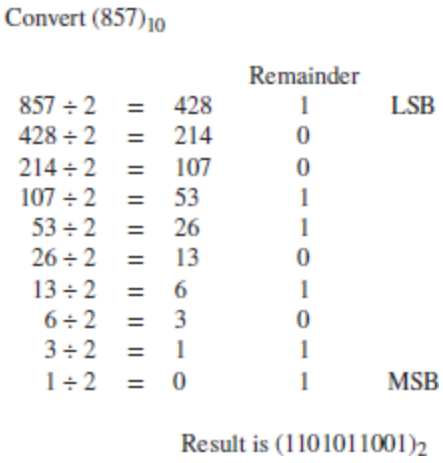


Figure 1.6 Conversion from decimal to binary.

So far, we have considered only the representation of positive integers. In Chapter 3 we will complete the discussion of number representation by explaining how negative numbers are handled and how fixed-point and floating-point numbers may be represented. We will also explain how arithmetic operations are performed in computers.

1.5.3 ASCII CHARACTER CODE

Alphanumeric information, such as letters and numbers typed on a computer keyboard, is represented as codes consisting of 0 and 1 digits. The most common code used for this type of information is known as the *ASCII code*, which stands for the American Standard Code for Information Interchange. The code specified by this standard is presented in Table 1.2.

Table 1-2 The seven-bit ASCII code.

Bit positions	Bit positions 654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

NUL	Null/Idle	SI	Shift in
SOH	Start of header	DLE	Data link escape
STX	Start of text	DC1-DC4	Device control
ETX	End of text	NAK	Negative acknowledgement
EOT	End of transmission	SYN	Synchronous idle
ENQ	Enquiry	ETB	End of transmitted block
ACQ	Acknowledgement	CAN	Cancel (error in data)
BEL	Audible signal	EM	End of medium
BS	Back space	SUB	Special sequence
HT	Horizontal tab	ESC	Escape
LF	Line feed	FS	File separator
VT	Vertical tab	GS	Group separator
FF	Form feed	RS	Record separator
CR	Carriage return	US	Unit separator
SO	Shift out	DEL	Delete/Idle

Bit positions of code format = | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

The ASCII code uses seven-bit patterns to denote 128 different characters. Ten of the characters are decimal digits 0 to 9. As the table shows, the high-order bits have the same pattern, $b_6b_5b_4 = 011$, for all 10 digits. Each digit is identified by the low-order four bits, b_{3-0}, using the binary patterns for these digits. Capital and lowercase letters are encoded in a way that makes sorting of textual information easy. The codes for A to Z are in ascending numerical sequence, which means that the task of sorting letters (or words) can be accomplished by a simple arithmetic comparison of the codes that represent the letters.

In addition to codes that represent characters and letters, the ASCII code includes punctuation marks such as ! and ?, commonly used symbols such as & and %, and a collection of control characters. The control characters are those needed in computer systems to handle and transfer data among various devices. For example, the carriage return character, which is abbreviated as CR in the table, indicates that the carriage, or cursor position, of an output device, such as a printer or display, should return to the leftmost column.

The ASCII code is used to encode information that is handled as text. It is not convenient for representation of numbers that are used as operands in arithmetic operations. For this purpose, it is best to convert ASCII-encoded numbers into a binary representation that we discussed before.

The ASCII standard uses seven bits to encode a character. In computer systems a more natural size is eight bits, or one byte. There are two common ways of fitting an ASCII-encoded character into a byte. One is to set the eighth bit, b_7, to 0. Another is to use this bit to indicate the *parity* of the other seven bits, which means showing whether the number of 1s in the seven-bit code is even or odd. We discuss parity in Chapter 4.

1.5.4 DIGITAL AND ANALOG INFORMATION

Binary numbers can be used to represent many types of information. For example, they can represent music that is stored in a personal music player. Figure 1.7 illustrates a music player, which contains an electronic memory for storing music files. A music file comprises a sequence of binary numbers that represent tones. To convert these binary numbers into sound, a *digital-to-analog (D/A) converter* circuit is used. It converts digital values into corresponding voltage levels, which create an analog voltage signal that drives the speakers inside the headphones. The binary values stored in the music player are referred to as *digital* information, whereas the voltage signal that drives the speakers is *analog* information.

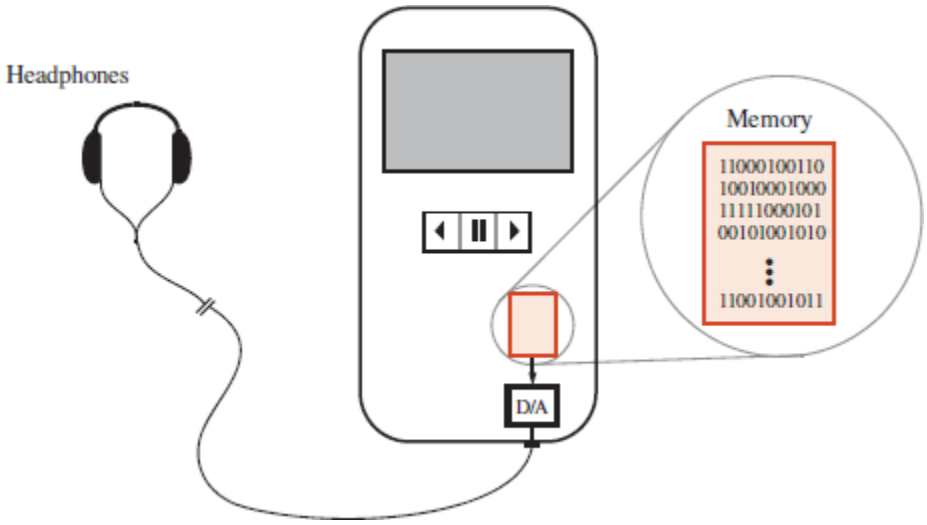


Figure 1.7 Using digital technology to represent music.

1.6 Theory and Practice

Modern design of logic circuits depends heavily on CAD tools, but the discipline of logic design evolved long before CAD tools were invented. This chronology is quite obvious because the very first computers were built with logic circuits, and there certainly were no computers available on which to design them!

Numerous manual design techniques have been developed to deal with logic circuits. Boolean algebra, which we will introduce in Chapter 2, was adopted as a mathematical means for representing such circuits. An enormous amount of "theory" was developed showing how certain design issues may be treated. To be successful, a designer had to apply this knowledge in practice.

CAD tools not only made it possible to design incredibly complex circuits but also made the design work much simpler in general. They perform many tasks automatically, which may suggest that today's designer need not understand the theoretical concepts used in the tasks performed by CAD tools. An obvious question would then be, Why should one study the theory that is no longer needed for manual design? Why not simply learn how to use the CAD tools?

There are three big reasons for learning the relevant theory. First, although the CAD tools perform the automatic tasks of optimizing a logic circuit to meet particular design objectives, the designer has to give the original description of the logic circuit. If the designer specifies a circuit that has inherently bad properties, then the final circuit will also be of poor quality. Second, the algebraic rules and theorems for design and manipulation of logic circuits are directly implemented in today's CAD tools. It is not possible for a user of the tools to understand what the tools do without grasping the underlying theory. Third, CAD tools offer many optional processing steps that a user can invoke when working on a design. The designer chooses which options to use by examining the resulting circuit produced by the CAD tools and deciding whether it meets the required objectives. The only way that the designer can know whether or not to apply a particular option in a given situation is to know what the CAD tools will do if that option is invoked—again, this implies that the designer must be familiar with the underlying theory. We discuss the logic circuit theory extensively in this book, because it is not possible to become an effective logic circuit designer without understanding the fundamental concepts.

There is another good reason to learn some logic circuit theory even if it were not required for CAD tools. Simply put, it is interesting and intellectually challenging. In the modern world filled with sophisticated automatic machinery, it is tempting to rely on tools as a substitute for thinking. However, in logic circuit design, as in any type of design process, computer-based tools are not a substitute for human intuition and innovation. Computer-based tools can produce good digital hardware designs only when employed by a designer who thoroughly understands the nature of logic circuits.

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

***1.1** Convert the following decimal numbers into binary, using the method shown in Figure 1.6

(a) $(20)_{10}$
(b) $(100)_{10}$
(c) $(129)_{10}$
(d) $(260)_{10}$
(e) $(10240)_{10}$

1.2 Convert the following decimal numbers into binary, using the method shown in Figure 1.6
(a) $(30)_{10}$
(b) $(110)_{10}$
(c) $(259)_{10}$
(d) $(500)_{10}$
(e) $(20480)_{10}$

1.3 Convert the following decimal numbers into binary, using the method shown in Figure 1.6
(a) $(1000)_{10}$
(b) $(10000)_{10}$
(c) $(100000)_{10}$
(d) $(1000000)_{10}$

***1.4** In Figure 1.6 we show how to convert a decimal number into binary by successively dividing by 2. Another way to derive the answer is to construct the number by using powers of 2. For example, if we wish to convert the number $(23)_{10}$, then the largest power of 2 that is not larger than 23 is $2^4 = 16$. Hence, the binary number will have five bits and the most-significant bit is $b_4 = 1$. We then perform the subtraction $23 - 16 = 7$. Now, the largest power of 2 that is not larger than 7 is $2^2 = 4$. Hence, $b_3 = 0$ (because $2^3 = 8$ is larger than 7) and $b_2 = 1$. Continuing this process gives

$$\begin{aligned} 23 &= 16 + 4 + 2 + 1 \\ &= 2^4 + 2^2 + 2^1 + 2^0 \\ &= 10000 + 00100 + 00010 + 00001 \\ &= 10111 \end{aligned}$$

Using this method, convert the following decimal numbers into binary.
(a) $(17)_{10}$
(b) $(33)_{10}$
(c) $(67)_{10}$
(d) $(130)_{10}$

(e) $(2560)_{10}$

(f) $(51200)_{10}$

1.5 Repeat Problem 1.3 using the method described in Problem 1.4.

***1.6** Convert the following binary numbers into decimal

(a) $(1001)_2$

(b) $(11100)_2$

(c) $(111111)_2$

(d) $(101010101010)_2$

1.7 Convert the following binary numbers into decimal

(a) $(110010)_2$

(b) $(1100100)_2$

(c) $(11001000)_2$

(d) $(110010000)_2$

***1.8** What is the minimum number of bits needed to represent the following decimal numbers in binary?

(a) $(270)_{10}$

(b) $(520)_{10}$

(c) $(780)_{10}$

(d) $(1029)_{10}$

1.9 Repeat Problem 1.8 for the following decimal numbers:

(a) $(111)_{10}$

(b) $(333)_{10}$

(c) $(555)_{10}$

(d) $(1111)_{10}$

chapter

2

INTRODUCTION TO LOGIC CIRCUITS

CHAPTER OBJECTIVES

In this chapter you will be introduced to:

- Logic functions and circuits
- Boolean algebra for dealing with logic functions
- Logic gates and synthesis of simple circuits
- CAD tools and the VHDL hardware description language
- Minimization of functions and Karnaugh maps

The study of logic circuits is motivated mostly by their use in digital computers. But such circuits also form the foundation of many other digital systems, such as those that perform control applications or are involved in digital communications. All such applications are based on some simple logical operations that are performed on input information.

In Chapter 1 we showed that information in computers is represented as electronic signals that can have two discrete values. Although these values are

implemented as voltage levels in a circuit, we refer to them simply as logic values, 0 and 1. Any circuit in which the signals are constrained to have only some number of discrete values is called a *logic circuit*. Logic circuits can be designed with different numbers of logic values, such as three, four, or even more, but in this book we deal only with the *binary* logic circuits that have two logic values.

Binary logic circuits have the dominant role in digital technology. We hope to provide the reader with an understanding of how these circuits work, how they are represented in mathematical notation, and how they are designed using modern design automation techniques. We begin by introducing some basic concepts pertinent to the binary logic circuits.

2.1 Variables and Functions

The dominance of binary circuits in digital systems is a consequence of their simplicity, which results from constraining the signals to assume only two possible values. The simplest binary element is a switch that has two states. If a given switch is controlled by an input variable x, then we will say that the switch is open if $x = 0$ and closed if $x = 1$, as illustrated in Figure 2.1*a*. We will use the graphical symbol in Figure 2.1*b* to represent such switches in the diagrams that follow. Note that the control input x is shown explicitly in the symbol. In Appendix B we explain how such switches are implemented with transistors.

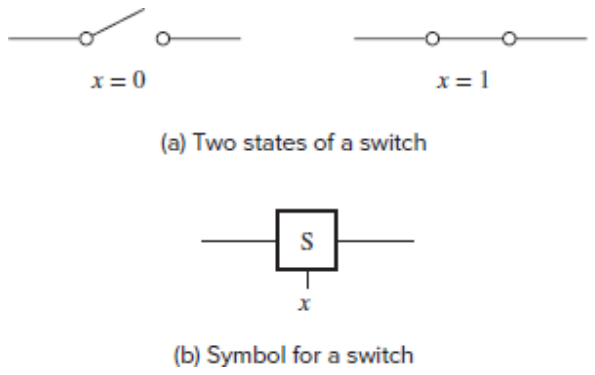


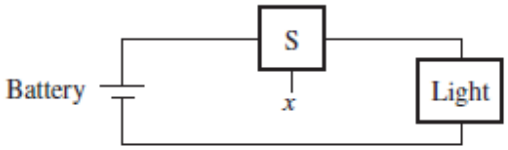
Figure 2.1 A binary switch.

Consider a simple application of a switch, where the switch turns a small lightbulb on or off. This action is accomplished with the circuit in Figure

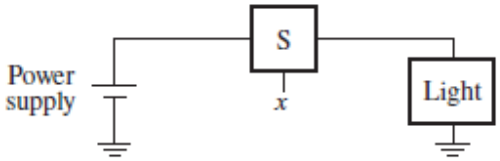
2.2*a*. A battery provides the power source. The lightbulb glows when a sufficient amount of current passes through it. The current flows when the switch is closed, that is, when $x = 1$. In this example the input that causes changes in the behavior of the circuit is the switch control x. The output is defined as the state (or condition) of the light, which we will denote by the letter L. If the light is on, we will say that $L = 1$. If the light is off, we will say that $L = 0$. Using this convention, we can describe the state of the light as a function of the input variable x. Since $L = 1$ if $x = 1$ and $L = 0$ if $x = 0$, we can say that

$$L(x) = x$$

This simple *logic expression* describes the output as a function of the input. We say that $L(x) = x$ is a *logic function* and that x is an *input variable*.



(a) Simple connection to a battery



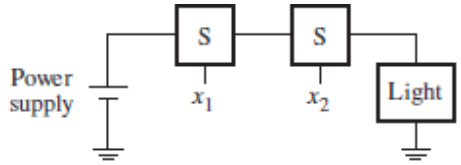
(b) Using a ground connection as the return path

Figure 2.2 A light controlled by a switch.

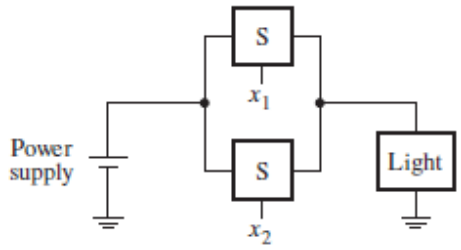
The circuit in Figure 2.2*a* can be found in an ordinary flashlight, where the switch is a simple mechanical device. In an electronic circuit the switch is implemented as a transistor and the light may be a light-emitting diode (LED). An electronic circuit is powered by a power supply of a certain voltage, usually in the range of 1 to 5 volts. One side of the power supply provides the *circuit ground*, as illustrated in Figure 2.2*b*. The circuit ground is a common reference point for voltages in the circuit. Rather than drawing wires in a circuit diagram for all nodes that return to the circuit ground, the

diagram can be simplified by showing a connection to a ground symbol, as we have done for the bottom terminal of the light L in the figure. In the circuit diagrams that follow we will use this convention, because it makes the diagrams look simpler.

Consider now the possibility of using two switches to control the state of the light. Let x_1 and x_2 be the control inputs for these switches. The switches can be connected either in series or in parallel as shown in Figure 2.3. Using a series connection, the light will be turned on only if both switches are closed. If either switch is open, the light will be off. This behavior can be described by the expression



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

Figure 2.3 Two basic functions.

$$L(x_1, x_2) = x_1 \cdot x_2$$

where

$L = 1$ if $x_1 = 1$ and $x_2 = 1$,

$L = 0$ otherwise.

The "$\cdot$" symbol is called the *AND operator*, and the circuit in Figure 2.3*a* is said to implement a *logical AND function*.

The parallel connection of two switches is given in Figure 2.3*b*. In this case the light will be on if either the x_1 or x_2 switch is closed. The light will also be on if both switches are closed. The light will be off only if both switches are open. This behavior can be stated as

$$L(x_1, x_2) = x_1 + x_2$$

where

$L = 1$ if $x_1 = 1$ or $x_2 = 1$ or if $x_1 = x_2 = 1$,

$L = 0$ if $x_1 = x_2 = 0$.

The + symbol is called the *OR operator*, and the circuit in Figure 2.3*b* is said to implement a *logical OR function*. It is important not to confuse the use of the + symbol with its more common meaning, which is for arithmetic addition. In this chapter the + symbol represents the logical OR operation unless otherwise stated. In the above expressions for AND and OR, the output $L(x_1, x_2)$ is a logic function with input variables x_1 and x_2. The AND and OR functions are two of the most important logic functions. Together with some other simple functions, they can be used as building blocks for the implementation of all logic circuits. Figure 2.4 illustrates how three switches can be used to control the light in a more complex way. This series-parallel connection of switches realizes the logic function

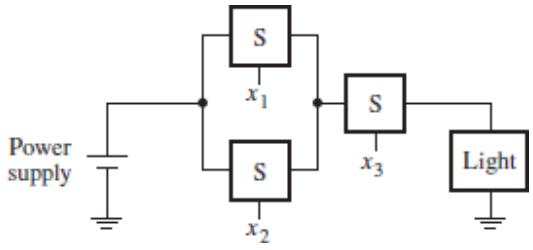


Figure 2.4 A series-parallel connection.

$$L(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$$

The light is on if $x_3 = 1$ and, at the same time, at least one of the x_1 or x_2 inputs is equal to 1.

2.2 INVERSION

So far we have assumed that some positive action takes place when a switch is closed, such as turning the light on. It is equally interesting and useful to consider the possibility that a positive action takes place when a switch is opened. Suppose that we connect the light as shown in Figure 2.5. In this case the switch is connected in parallel with the light, rather than in series. Consequently, a closed switch will short-circuit the light and prevent the current from flowing through it. Note that we have included an extra resistor in this circuit to ensure that the closed switch does not short-circuit the power

supply. The light will be turned on when the switch is opened. Formally, we express this functional behavior as

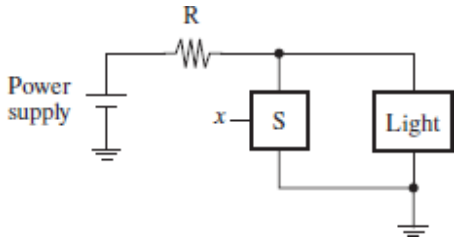


Figure 2.5 An inverting circuit.

$$L(x) = \bar{x}$$
$$\text{where} \quad L = 1 \text{ if } x = 0,$$
$$L = 0 \text{ if } x = 1$$

The value of this function is the inverse of the value of the input variable. Instead of using the word *inverse*, it is more common to use the term *complement*. Thus we say that $L(x)$ is a complement of x in this example. Another frequently used term for the same operation is the *NOT operation*. There are several commonly used notations for indicating the complementation. In the preceding expression, we placed an overbar on top of x. This notation is probably the best from the visual point of view. However, when complements are needed in expressions that are typed using a computer keyboard, which is often done when using CAD tools, it is impractical to use overbars. Instead, either an apostrophe is placed after the variable, or an exclamation mark (!), the tilde character (~), or the word NOT is placed in front of the variable to denote the complementation. Thus the following are equivalent:

$$\bar{x} = x' = !x = \sim x = \text{NOT } x$$

The complement operation can be applied to a single variable or to more complex operations. For example, if

$$f(x_1, x_2) = x_1 + x_2$$

then the complement of f is

$$\bar{f}(x_1, x_2) = \overline{x_1 + x_2}$$

This expression yields the logic value 1 only when neither x_1 nor x_2 is equal to 1, that is, when $x_1 = x_2 = 0$. Again, the following notations are equivalent:

$$\overline{x_1 + x_2} = (x_1 + x_2)' = !(x_1 + x_2) = \sim(x_1 + x_2) = \text{NOT } (x_1 + x_2)$$

2.3 Truth Tables

We have introduced the three most basic logic operations—AND, OR, and complement—by relating them to simple circuits built with switches. This approach gives these operations a certain "physical meaning." The same operations can also be defined in the form of a table, called a *truth table*, as shown in Figure 2.6. The first two columns (to the left of the double vertical line) give all four possible combinations of logic values that the variables x_1 and x_2 can have. The next column defines the AND operation for each combination of values of x_1 and x_2, and the last column defines the OR operation. Because we will frequently need to refer to "combinations of logic values" applied to some variables, we will adopt a shorter term, *valuation*, to denote such a combination of logic values.

x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
		AND	OR

Figure 2.6 A truth table for the AND and OR operations.

The truth table is a useful aid for depicting information involving logic functions. We will use it in this book to define specific functions and to show the validity of certain functional relations. Small truth tables are easy to deal with. However, they grow exponentially in size with the number of variables. A truth table for three input variables has eight rows because there are eight possible valuations of these variables. Such a table is given in Figure 2.7, which defines three-input AND and OR functions. For four input variables the

truth table has 16 rows, and so on. In general, for n input variables the truth table has 2^n rows.

x_1	x_2	x_3	$x_1 \cdot x_2 \cdot x_3$	$x_1 + x_2 + x_3$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 2.7 Three-input AND and OR operations.

The AND and OR operations can be extended to n variables. An AND function of variables $x_1, x_2, \ldots, x_n$ has the value 1 only if all n variables are equal to 1. An OR function of variables $x_1, x_2, \ldots, x_n$ has the value 1 if one or more of the variables is equal to 1.

2.4 Logic Gates and Networks

The three basic logic operations introduced in the previous sections can be used to implement logic functions of any complexity. A complex function may require many of these basic operations for its implementation. Each logic operation can be implemented electronically with transistors, resulting in a circuit element called a *logic gate*. A logic gate has one or more inputs and one output that is a function of its inputs. It is often convenient to describe a logic circuit by drawing a circuit diagram, or *schematic*, consisting of graphical symbols representing the logic gates. The graphical symbols for the AND, OR, and NOT gates are shown in Figure 2.8. The figure indicates on the left side how the AND and OR gates are drawn when there are only a few inputs. On the right side it shows how the symbols are augmented to accommodate a greater number of inputs. We show how logic gates are built using transistors in Appendix B.

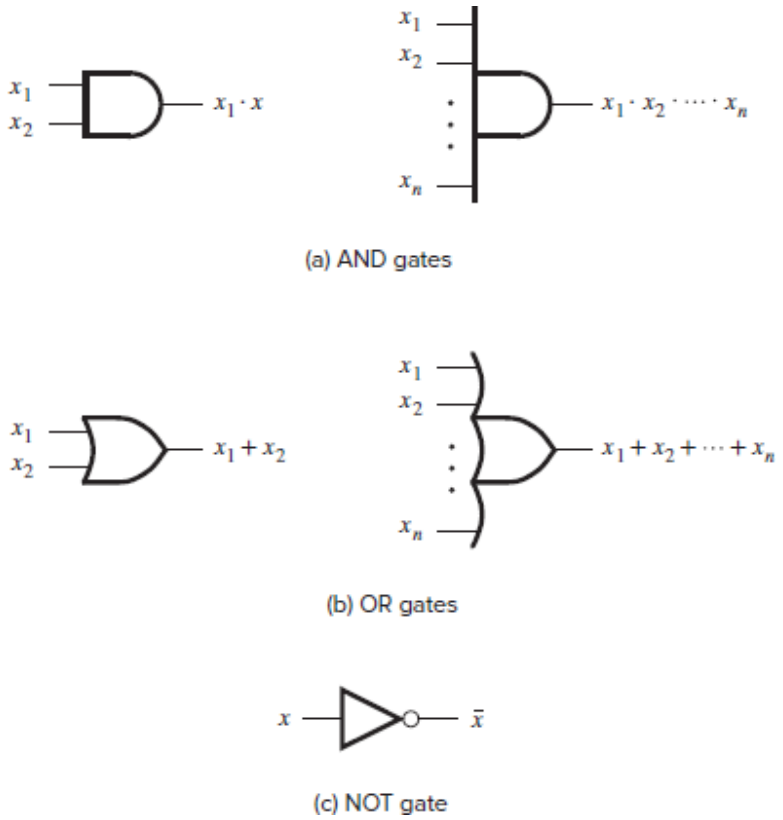


Figure 2.8 The basic gates.

A larger circuit is implemented by a *network* of gates. For example, the logic function from Figure 2.4 can be implemented by the network in Figure 2.9. The complexity of a given network has a direct impact on its cost. Because it is always desirable to reduce the cost of any manufactured product, it is important to find ways for implementing logic circuits as inexpensively as possible. We will see shortly that a given logic function can be implemented with a number of different networks. Some of these networks are simpler than others; hence, searching for the solutions that entail minimum cost is prudent.

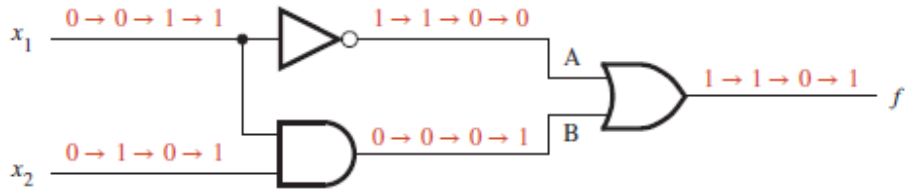
Figure 2.9 The function from Figure 2.4.

In technical jargon, a network of gates is often called a *logic network* or simply a *logic circuit*. We will use these terms interchangeably.

2.4.1 ANALYSIS OF A LOGIC NETWORK

A designer of digital systems is faced with two basic issues. For an existing logic network, it must be possible to determine the function performed by the network. This task is referred to as the *analysis* process. The reverse task of designing a new network that implements a desired functional behavior is referred to as the *synthesis* process. The analysis process is rather straightforward and much simpler than the synthesis process.

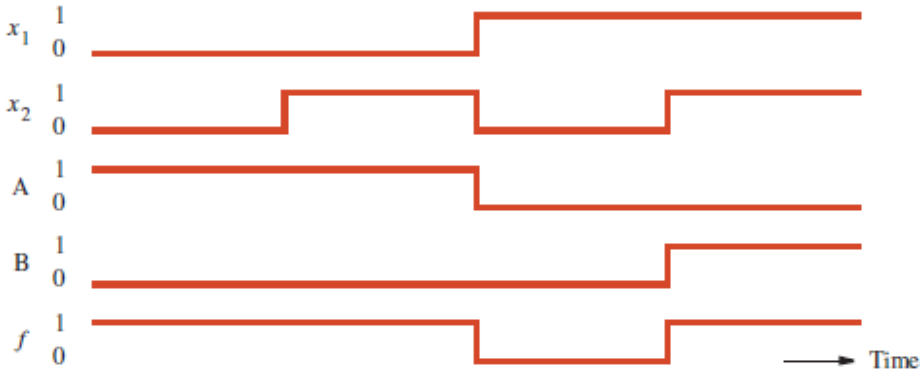
Figure 2.10*a* shows a simple network consisting of three gates. To analyze its functional behavior, we can consider what happens if we apply all possible combinations of the input signals to it. Suppose that we start by making $x_1 = x_2 = 0$. This forces the output of the NOT gate to be equal to 1 and the output of the AND gate to be 0. Because one of the inputs to the OR gate is 1, the output of this gate will be 1. Therefore, $f = 1$ if $x_1 = x_2 = 0$. If we then let $x_1 = 0$ and $x_2 = 1$, no change in the value of f will take place, because the outputs of the NOT and AND gates will still be 1 and 0, respectively. Next, if we apply $x_1 = 1$ and $x_2 = 0$, then the output of the NOT gate changes to 0 while the output of the AND gate remains at 0. Both inputs to the OR gate are then equal to 0; hence the value of f will be 0. Finally, let $x_1 = x_2 = 1$. Then the output of the AND gate goes to 1, which in turn causes f to be equal to 1. Our verbal explanation can be summarized in the form of the truth table shown in Figure 2.10*b*.


x_1
$0 \rightarrow 0 \rightarrow 1 \rightarrow 1$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 0$
A
$1 \rightarrow 1 \rightarrow 0 \rightarrow 1$
f
$0 \rightarrow 0 \rightarrow 0 \rightarrow 1$
B
x_2
$0 \rightarrow 1 \rightarrow 0 \rightarrow 1$

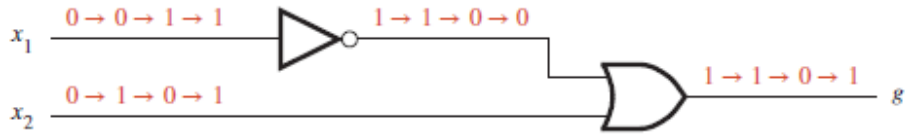
(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

x_1	x_2	$f(x_1, x_2)$	A	B
0	0	1	1	0
0	1	1	1	0
1	0	0	0	0
1	1	1	0	1

(b) Truth table



(c) Timing diagram


x_1
$0 \rightarrow 0 \rightarrow 1 \rightarrow 1$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 0$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 1$
g
x_2
$0 \rightarrow 1 \rightarrow 0 \rightarrow 1$

(d) Network that implements $g = \bar{x}_1 + x_2$

Figure 2.10 An example of logic networks.

Timing Diagram

We have determined the behavior of the network in Figure 2.10*a* by considering the four possible valuations of the inputs x_1 and x_2. Suppose that the signals that correspond to these valuations are applied to the network in the order of our discussion; that is, (x_1, x_2) = (0, 0) followed by (0, 1), (1, 0), and (1, 1). Then changes in the signals at various points in the network would be as indicated in red in the figure. The same information can be presented in

graphical form, known as a *timing diagram*, as shown in Figure 2.10*c*. The time runs from left to right, and each input valuation is held for some fixed duration. The figure shows the waveforms for the inputs and output of the network, as well as for the internal signals at the points labeled *A* and *B*.

The timing diagram in Figure 2.10*c* shows that changes in the waveforms at points *A* and *B* and the output *f* take place instantaneously when the inputs x_1 and x_2 change their values. These idealized waveforms are based on the assumption that logic gates respond to changes on their inputs in zero time. Such timing diagrams are useful for indicating the *functional behavior* of logic circuits. However, practical logic gates are implemented using electronic circuits which need some time to change their states. Thus, there is a delay between a change in input values and a corresponding change in the output value of a gate. In chapters that follow we will use timing diagrams that incorporate such delays.

Timing diagrams are used for many purposes. They depict the behavior of a logic circuit in a form that can be observed when the circuit is tested using instruments such as logic analyzers and oscilloscopes. Also, they are often generated by CAD tools to show the designer how a given circuit is expected to behave before it is actually implemented electronically. We will introduce the CAD tools later in this chapter and will make use of them throughout the book.

Functionally Equivalent Networks

Now consider the network in Figure 2.10*d*. Going through the same analysis procedure, we find that the output *g* changes in exactly the same way as *f* does in part (*a*) of the figure. Therefore, $g(x_1, x_2) = f(x_1, x_2)$, which indicates that the two networks are functionally equivalent; the output behavior of both networks is represented by the truth table in Figure 2.10*b*. Since both networks realize the same function, it makes sense to use the simpler one, which is less costly to implement.

In general, a logic function can be implemented with a variety of different networks, probably having different costs. This raises an important question: How does one find the best implementation for a given function? We will discuss some of the main approaches for synthesizing logic functions later in this chapter. For now, we should note that some manipulation is needed to transform the more complex network in Figure 2.10*a* into the network in

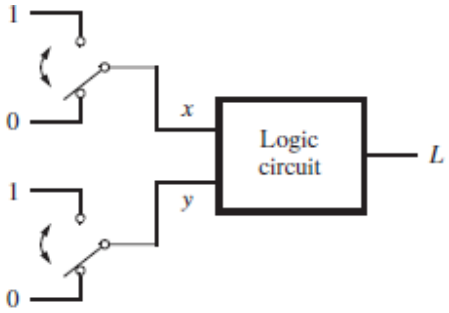
Figure 2.10*d*. Since $f(x_1,x_2) = \bar{x}_1 + x_1 \cdot x_2$ and $g(x_1,x_2) = \bar{x}_1 + x_2$, there must exist some rules that can be used to show the equivalence

$$\bar{x}_1 + x_1 \cdot x_2 = \bar{x}_1 + x_2$$

We have already established this equivalence through detailed analysis of the two circuits and construction of the truth table. But the same outcome can be achieved through algebraic manipulation of logic expressions. In Section 2.5 we will introduce a mathematical approach for dealing with logic functions, which provides the basis for modern design techniques.

Example 2.1

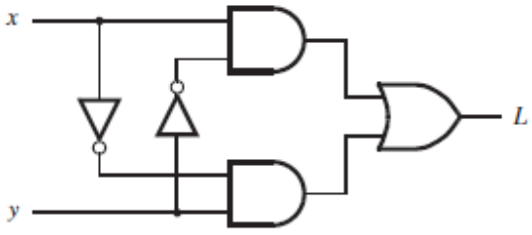
As an example of a logic function, consider the diagram in Figure 2.11*a*. It includes two toggle switches that control the values of signals *x* and *y*. Each toggle switch can be pushed down to the bottom position or up to the top position. When a toggle switch is in the bottom position it makes a connection to logic value 0 (ground), and when in the top position it connects to logic value 1 (power supply level). Thus, these toggle switches can be used to set *x* and *y* to either 0 or 1.



(a) Two switches that control a light

x	*y*	*L*
0	0	0
0	1	1
1	0	1
1	1	0

(b) Truth table



(c) Logic network

(d) XOR gate symbol

Figure 2.11 An example of a logic circuit.

The signals x and y are inputs to a logic circuit that controls a light L. The required behavior is that the light should be on only if one, but not both, of the toggle switches is in the top position. This specification leads to the truth table in part (*b*) of the figure. Since $L = 1$ when x=0 and y=1 or when x=1 and y=0, we can implement this logic function using the network in Figure 2.11*c*.

The reader may recognize the behavior of our light as being similar to that over a set of stairs in a house, where the light is controlled by two switches: one at the top of the stairs, and the other at the bottom. The light can be turned on or off by either switch because it follows the truth table in Figure 2.11*b*. This logic function, which differs from the OR function only when both inputs are equal to 1, is useful for other applications as well. It is called the *exclusive-OR* (XOR) function and is indicated in logic expressions by the symbol $\oplus$. Thus, rather than writing $L = \bar{x} \cdot y + x \cdot \bar{y}$, we can write $L = x \oplus y$. The XOR function has the logic-gate symbol illustrated in Figure 2.11*d*.

Example 2.2

In Chapter 1 we showed how numbers are represented in computers by using binary digits. As another example of logic functions, consider the addition of two one-digit binary numbers a and b. The four possible valuations of a,b and the resulting sums are given in Figure 2.12*a* (in this figure the + operator signifies *addition*). The sum $S = s_1 s_0$ has to be a two-digit binary number, because when $a = b = 1$ then $S = 10$.

Figure 2.12*b* gives a truth table for the logic functions s_1 and s_0. From this table we can see that $s_1 = a \cdot b$ and $s_0 = a \oplus b$. The corresponding logic network is given in part (*c*) of the figure. This type of logic circuit, which adds binary numbers, is referred to as an *adder* circuit. We discuss circuits of this type in Chapter 3.

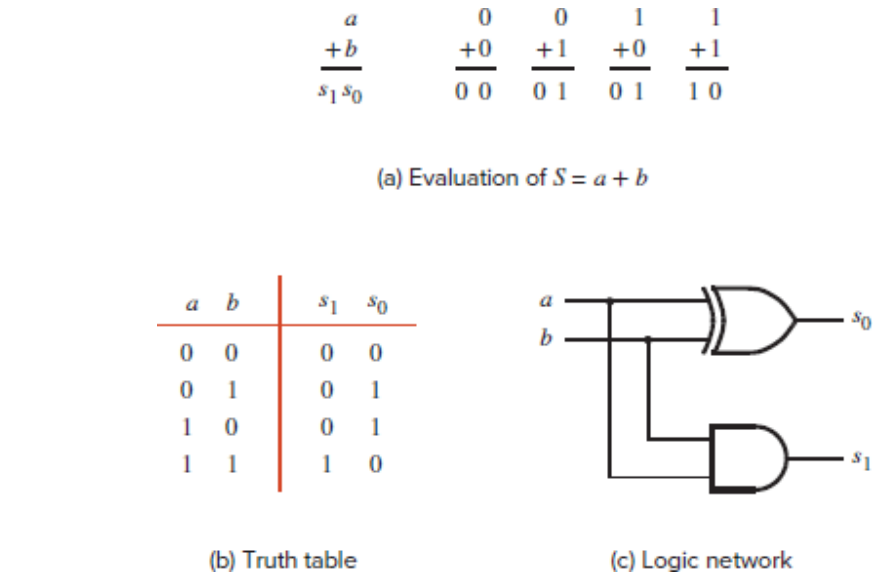


Figure 2.12 Addition of binary numbers.

2.5 Boolean Algebra

In 1849 George Boole published a scheme for the algebraic description of processes involved in logical thought and reasoning [1]. Subsequently, this scheme and its further refinements became known as *Boolean algebra*. It was almost 100 years later that this algebra found application in the engineering sense. In the late 1930s, Claude Shannon showed that Boolean algebra provides an effective means of describing circuits built with switches [2]. The algebra can, therefore, be used to describe logic circuits. We will show that this algebra is a powerful tool that can be used for designing and analyzing logic circuits. The reader will come to appreciate that it provides the foundation for much of our modern digital technology.

Axioms of Boolean Algebra

Like any algebra, Boolean algebra is based on a set of rules that are derived from a small number of basic assumptions. These assumptions are called *axioms*. Let us assume that Boolean algebra involves elements that take on one of two values, 0 and 1. Assume that the following axioms are true:

1*a*. $0 \cdot 0 = 0$
1*b*. $1 + 1 = 1$
2*a*. $1 \cdot 1 = 1$
2*b*. $0 + 0 = 0$
3*a*. $0 \cdot 1 = 1 \cdot 0 = 0$
3*b*. $1 + 0 = 0 + 1 = 1$
4*a*. If $x = 0$, then $\overline{x} = 1$
4*b*. If $x = 1$, then $\overline{x} = 0$

Single-Variable Theorems

From the axioms we can define some rules for dealing with single variables. These rules are often called *theorems*. If x is a Boolean variable, then the following theorems hold:

5*a*. $x \cdot 0 = 0$
5*b*. $x + 1 = 1$
6*a*. $x \cdot 1 = x$
6*b*. $x + 0 = x$
7*a*. $x \cdot x = x$
7*b*. $x + x = x$
8*a*. $x \cdot \overline{x} = 0$
8*b*. $x + \overline{x} = 1$
9. $\overline{\overline{x}} = x$

It is easy to prove the validity of these theorems by perfect induction, that is, by substituting the values $x = 0$ and $x = 1$ into the expressions and using the axioms given above. For example, in theorem 5*a*, if $x = 0$, then the theorem states that $0 \cdot 0 = 0$, which is true according to axiom 1*a*. Similarly, if $x = 1$, then theorem 5*a* states that $1 \cdot 0 = 0$, which is also true according to axiom 3*a*. The reader should verify that theorems 5*a* to 9 can be proven in this way.

Duality

Notice that we have listed the axioms and the single-variable theorems in pairs. This is done to reflect the important *principle of duality*. Given a logic expression, its *dual* is obtained by replacing all + operators with · operators, and vice versa, and by replacing all 0s with 1s, and vice versa. The dual of any true statement (axiom or theorem) in Boolean algebra is also a true statement. At this point in the discussion, the reader might not appreciate why duality is a useful concept. However, this concept will become clear later in the chapter, when we will show that duality implies that at least two different ways exist to express every logic function with Boolean algebra. Often, one

expression leads to a simpler physical implementation than the other and is thus preferable.

Two- and Three-Variable Properties

To enable us to deal with a number of variables, it is useful to define some two- and three-variable algebraic identities. For each identity, its dual version is also given. These identities are often referred to as *properties*. They are known by the names indicated below. If x, y, and z are Boolean variables, then the following properties hold:

10*a*.	$x \cdot y = y \cdot x$	*Commutative*
10*b*.	$x + y = y + x$	
11*a*.	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	*Associative*
11*b*.	$x + (y + z) = (x + y) + z$	
12*a*.	$x \cdot (y + z) = x \cdot y + x \cdot z$	*Distributive*
12*b*.	$x + y \cdot z = (x + y) \cdot (x + z)$	
13*a*.	$x + x \cdot y = x$	*Absorption*
13*b*.	$x \cdot (x + y) = x$	
14*a*.	$x \cdot y + x \cdot \overline{y} = x$	*Combining*
14*b*.	$(x + y) \cdot (x + \overline{y}) = x$	
15*a*.	$\overline{x \cdot y} = \overline{x} + \overline{y}$	*DeMorgan's theorem*
15*b*.	$\overline{x + y} = \overline{x} \cdot \overline{y}$	
16*a*.	$x + \overline{x} \cdot y = x + y$	
16*b*.	$x \cdot (\overline{x} + y) = x \cdot y$	
17*a*.	$x \cdot y + y \cdot z + \overline{x} \cdot z = x \cdot y + \overline{x} \cdot z$	*Consensus*
17*b*.	$(x + y) \cdot (y + z) \cdot (\overline{x} + z) = (x + y) \cdot (\overline{x} + z)$	

Again, we can prove the validity of these properties either by perfect induction or by performing algebraic manipulation. Figure 2.13 illustrates how perfect induction can be used to prove DeMorgan's theorem, using the format of a truth table. The evaluation of left-hand and right-hand sides of the identity in 15*a* gives the same result.

x	y	$x \cdot y$	$\overline{x \cdot y}$	$\overline{x}$	$\overline{y}$	$\overline{x} + \overline{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0
		LHS		RHS		

Figure 2.13 Proof of DeMorgan's theorem in 15*a*.

We have listed a number of axioms, theorems, and properties. Not all of these are necessary to define Boolean algebra. For example, assuming that the + and · operations are defined, it is sufficient to include theorems 5 and 8 and properties 10 and 12. These are sometimes referred to as Huntington's basic postulates [3]. The other identities can be derived from these postulates.

The preceding axioms, theorems, and properties provide the information necessary for performing algebraic manipulation of more complex expressions.

Example 2.3

Let us prove the validity of the logic equation

$$(x_1 + x_3) \cdot (\bar{x}_1 + \bar{x}_3) = x_1 \cdot \bar{x}_3 + \bar{x}_1 \cdot x_3$$

The left-hand side can be manipulated as follows. Using the distributive property, 12*a*, gives

$$\text{LHS} = (x_1 + x_3) \cdot \bar{x}_1 + (x_1 + x_3) \cdot \bar{x}_3$$

Applying the distributive property again yields

$$\text{LHS} = x_1 \cdot \bar{x}_1 + x_3 \cdot \bar{x}_1 + x_1 \cdot \bar{x}_3 + x_3 \cdot \bar{x}_3$$

Note that the distributive property allows ANDing the terms in parenthesis in a way analogous to multiplication in ordinary algebra. Next, according to theorem 8*a*, the terms $x_1 \cdot \bar{x}_1$ and $x_3 \cdot \bar{x}_3$ are both equal to 0. Therefore,

$$\text{LHS} = 0 + x_3 \cdot \bar{x}_1 + x_1 \cdot \bar{x}_3 + 0$$

From 6*b* it follows that

$$\text{LHS} = x_3 \cdot \bar{x}_1 + x_1 \cdot \bar{x}_3$$

Finally, using the commutative property, 10*a* and 10*b*, this becomes

$$\text{LHS} = x_1 \cdot \bar{x}_3 + \bar{x}_1 \cdot x_3$$

which is the same as the right-hand side of the initial equation.

Example 2.4

Consider the logic equation

$$x_1 \cdot \bar{x}_3 + \bar{x}_2 \cdot \bar{x}_3 + x_1 \cdot x_3 + \bar{x}_2 \cdot x_3 = \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot x_2 + x_1 \cdot \bar{x}_2$$

The left-hand side can be manipulated as follows

$$\begin{aligned} \text{LHS} &= x_1 \cdot \bar{x}_3 + x_1 \cdot x_3 + \bar{x}_2 \cdot \bar{x}_3 + \bar{x}_2 \cdot x_3 && \text{using } 10b \\ &= x_1 \cdot (\bar{x}_3 + x_3) + \bar{x}_2 \cdot (\bar{x}_3 + x_3) && \text{using } 12a \\ &= x_1 \cdot 1 + \bar{x}_2 \cdot 1 && \text{using } 8b \\ &= x_1 + \bar{x}_2 && \text{using } 6a \end{aligned}$$

The right-hand side can be manipulated as

$$\begin{aligned} \text{RHS} &= \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot (x_2 + \bar{x}_2) && \text{using } 12a \\ &= \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot 1 && \text{using } 8b \\ &= \bar{x}_1 \cdot \bar{x}_2 + x_1 && \text{using } 6a \\ &= x_1 + \bar{x}_1 \cdot \bar{x}_2 && \text{using } 10b \\ &= x_1 + \bar{x}_2 && \text{using } 16a \end{aligned}$$

Being able to manipulate both sides of the initial equation into identical expressions establishes the validity of the equation. Note that the same logic function is represented by either the left- or the right-hand side of the above equation, namely

$$\begin{aligned} f(x_1, x_2, x_3) &= x_1 \cdot \bar{x}_3 + \bar{x}_2 \cdot \bar{x}_3 + x_1 \cdot x_3 + \bar{x}_2 \cdot x_3 \\ &= \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot x_2 + x_1 \cdot \bar{x}_2 \end{aligned}$$

As a result of manipulation, we have found a much simpler expression

$$f(x_1, x_2, x_3) = x_1 + \bar{x}_2$$

which also represents the same function. This simpler expression would result in a lower-cost logic circuit that could be used to implement the function.

Examples 2.3 and 2.4 illustrate the purpose of the axioms, theorems, and properties as a mechanism for algebraic manipulation. Even these simple

examples suggest that it is impractical to deal with highly complex expressions in this way. However, these theorems and properties provide the basis for automating the synthesis of logic functions in CAD tools. To understand what can be achieved using these tools, the designer needs to be aware of the fundamental concepts.

2.5.1 THE VENN DIAGRAM

We have suggested that perfect induction can be used to verify the theorems and properties. This procedure is quite tedious and not very informative from the conceptual point of view. A simple visual aid that can be used for this purpose also exists. It is called the Venn diagram, and the reader is likely to find that it provides for a more intuitive understanding of how two expressions may be equivalent.

The Venn diagram has traditionally been used in mathematics to provide a graphical illustration of various operations and relations in the algebra of sets. A set s is a collection of elements that are said to be the members of s. In the Venn diagram the elements of a set are represented by the area enclosed by a contour such as a square, a circle, or an ellipse. For example, in a universe N of integers from 1 to 10, the set of even numbers is $E = \{2, 4, 6, 8, 10\}$. A contour representing E encloses the even numbers. The odd numbers form the complement of E; hence the area outside the contour represents $\overline{E} = \{1, 3, 5, 7, 9\}$.

Since in Boolean algebra there are only two values (elements) in the universe, $B = \{0, 1\}$, we will say that the area within a contour corresponding to a set s denotes that $s = 1$, while the area outside the contour denotes $s = 0$. In the diagram we will shade the area where $s = 1$. The concept of the Venn diagram is illustrated in Figure 2.14. The universe B is represented by a square. Then the constants 1 and 0 are represented as shown in parts (a) and (b) of the figure. A variable, say, x, is represented by a circle, such that the area inside the circle corresponds to $x = 1$, while the area outside the circle corresponds to $x = 0$. This is illustrated in part (c). An expression involving one or more variables is depicted by shading the area where the value of the expression is equal to 1. Part (d) indicates how the complement of x is represented.

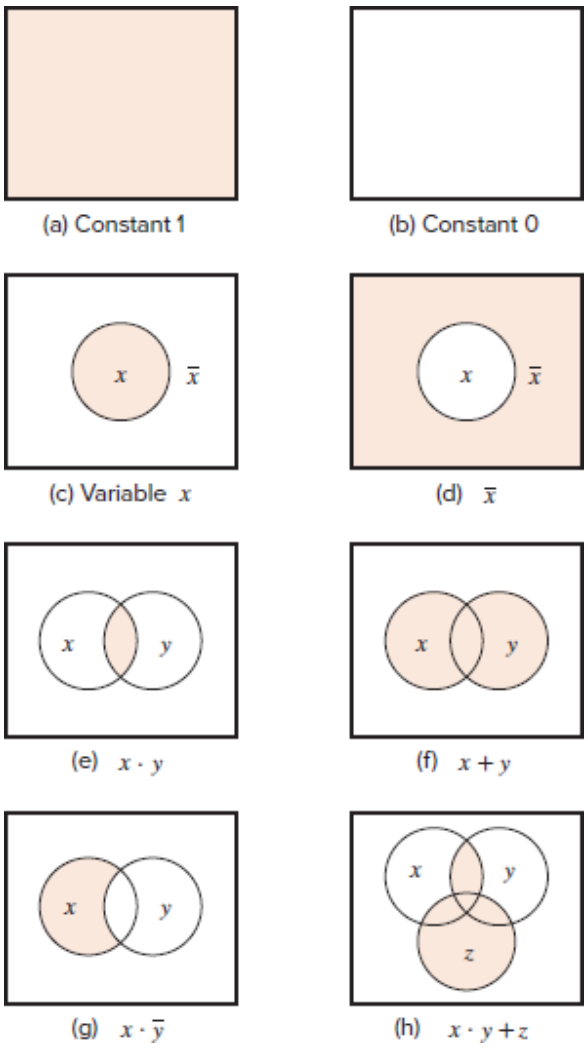


Figure 2.14 The Venn diagram representation.

To represent two variables, x and y, we draw two overlapping circles. Then the area where the circles overlap represents the case where $x = y = 1$, namely, the AND of x and y, as shown in part (*e*). Since this common area consists of the intersecting portions of x and y, the AND operation is often referred to formally as the *intersection* of x and y. Part (*f*) illustrates the OR operation, where $x + y$ represents the total area within both circles, namely, where at least one of x or y is equal to 1. Since this combines the areas in the circles, the OR operation is formally often called the *union* of x and y.

Part (*g*) depicts the term $x \cdot \bar{y}$, which is represented by the intersection of the area for x with that for $\bar{y}$. Part (*h*) gives a three-variable example; the expression $x \cdot y + z$ is the union of the area for z with that of the intersection of x and y.

To see how we can use Venn diagrams to verify the equivalence of two expressions, let us demonstrate the validity of the distributive property, 12*a*, in Section 2.5. Figure 2.15 gives the construction of the left and right sides of the identity that defines the property

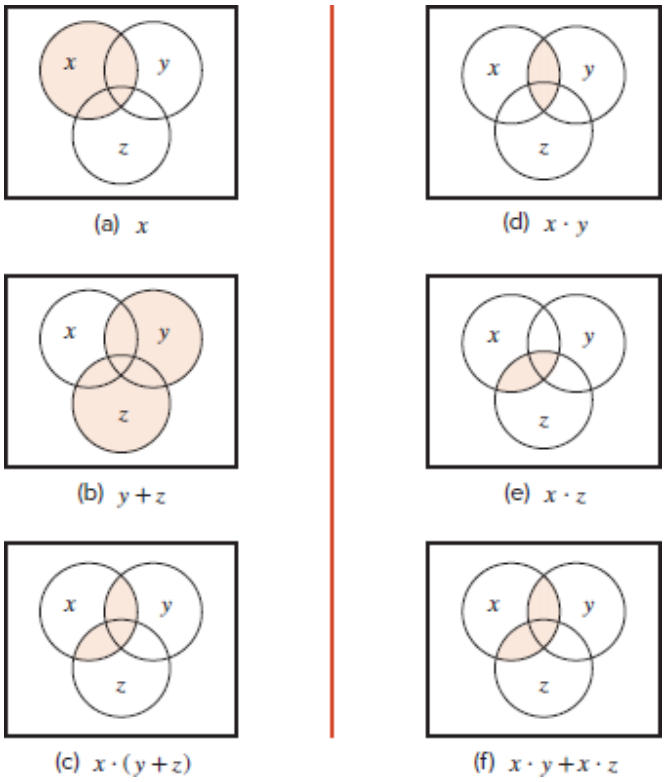


Figure 2.15 Verification of the distributive property $x \cdot (y + z) = x \cdot y + x \cdot z$.

$$x \cdot (y + z) = x \cdot y + x \cdot z$$

Part (*a*) shows the area where $x = 1$. Part (*b*) indicates the area for $y + z$. Part (*c*) gives the diagram for $x \cdot (y + z)$, the intersection of shaded areas in parts (*a*) and (*b*). The right-hand side is constructed in parts (*d*), (*e*), and (*f*). Parts (*d*) and (*e*) describe the terms $x \cdot y$ and $x \cdot z$, respectively. The union of the shaded areas in these two diagrams then corresponds to the expression $x \cdot y + x \cdot z$, as seen in part (*f*). Since the shaded areas in parts (*c*) and (*f*) are identical, it follows that the distributive property is valid.

As another example, consider the identity

$$x \cdot y + \bar{x} \cdot z + y \cdot z = x \cdot y + \bar{x} \cdot z$$

which is illustrated in Figure 2.16. Notice that this identity states that the term $y \cdot z$ is fully covered by the terms $x \cdot y$ and $\bar{x} \cdot z$; therefore, this term can be omitted. This identity, which we listed earlier as property 17*a*, is often referred to as *consensus*.

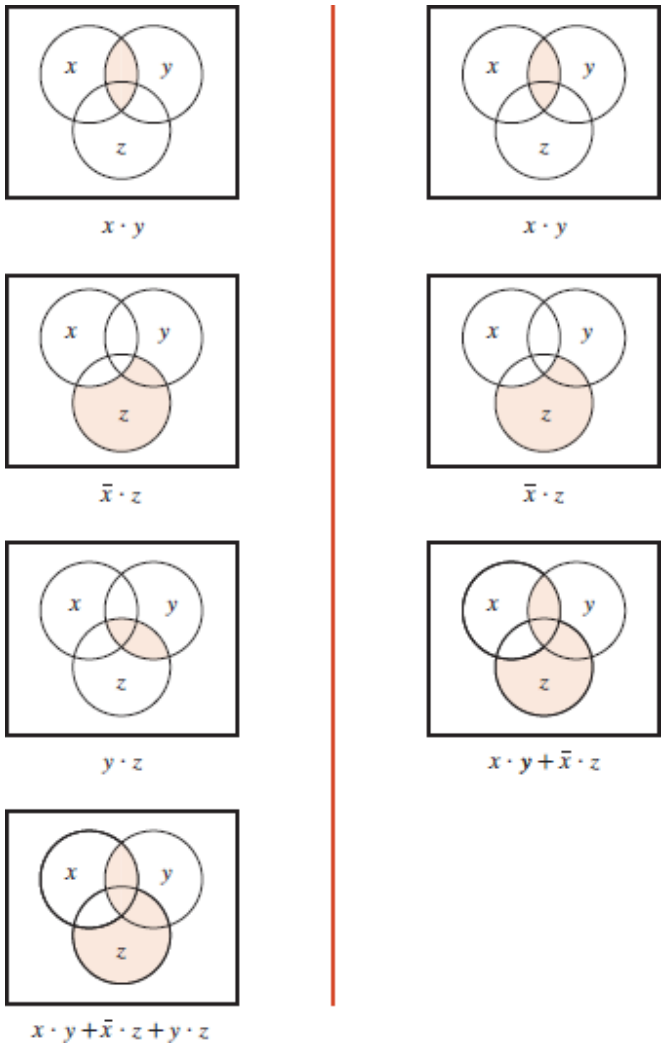


Figure 2.16 Verification of $x \cdot y + \bar{x} \cdot z + y \cdot z = x \cdot y + \bar{x} \cdot z$.

The reader should use the Venn diagram to prove some other identities. The examples below prove the distributive property 12*b*, and DeMorgan's theorem, 15*a*.

Example 2.5

The distributive property 12*a* in Figure 2.15 will look familiar to the reader, because it is valid both for Boolean variables and for variables that are real numbers. In the case of real-number variables, the operations involved would be multiplication and addition, rather than logical AND and OR. However, the dual form 12*b* of this property, $x + y \cdot z = (x + y) \cdot (x + z)$, does not hold for real-number variables involving multiplication and addition operations. To prove that this identity is valid in Boolean algebra, we can use the Venn diagrams in Figure 2.17. Parts (*a*) and (*b*) of the figure depict the terms x and $y \cdot z$, respectively, and part (*c*) gives the union of parts (*a*) and (*b*). Parts (*d*) and (*e*) depict the sum terms $(x + y)$ and $(x + z)$, and part (*f*) shows the intersection of (*d*) and (*e*). Since the diagrams in (*c*) and (*f*) are the same, this proves the identity.

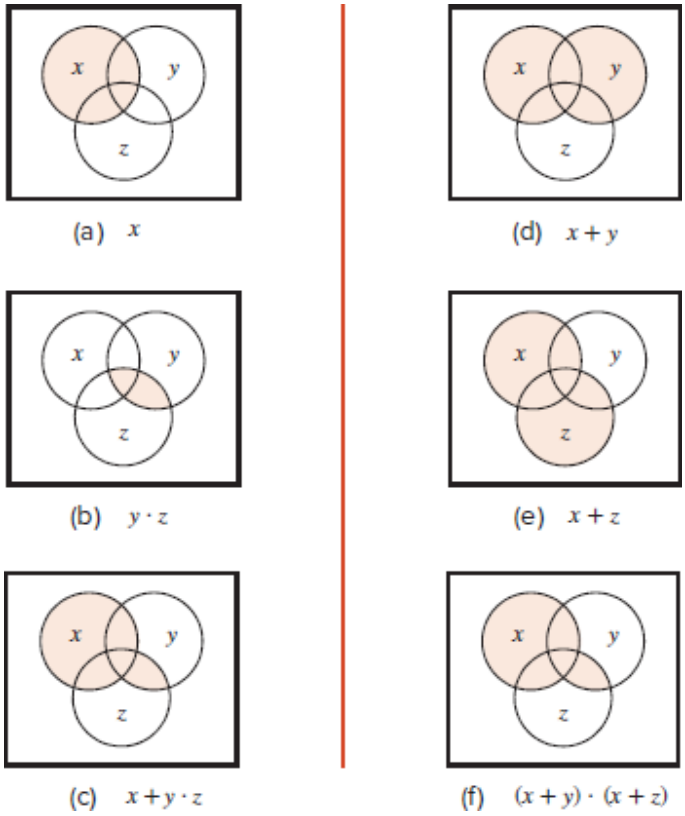


Figure 2.17 Proof of the distributive property 12*b*.

Example 2.6

A proof of DeMorgan's theorem 15*a* by using Venn diagrams is illustrated in Figure 2.18. The diagram in part (*b*) of the figure, which is the complement of $x \cdot y$, is the same as the diagram in part (*e*), which is the union of part (*c*) with part (*d*), thus proving the theorem. We leave it as an exercise for the reader to prove the dual form of DeMorgan's theorem, 15*b*.

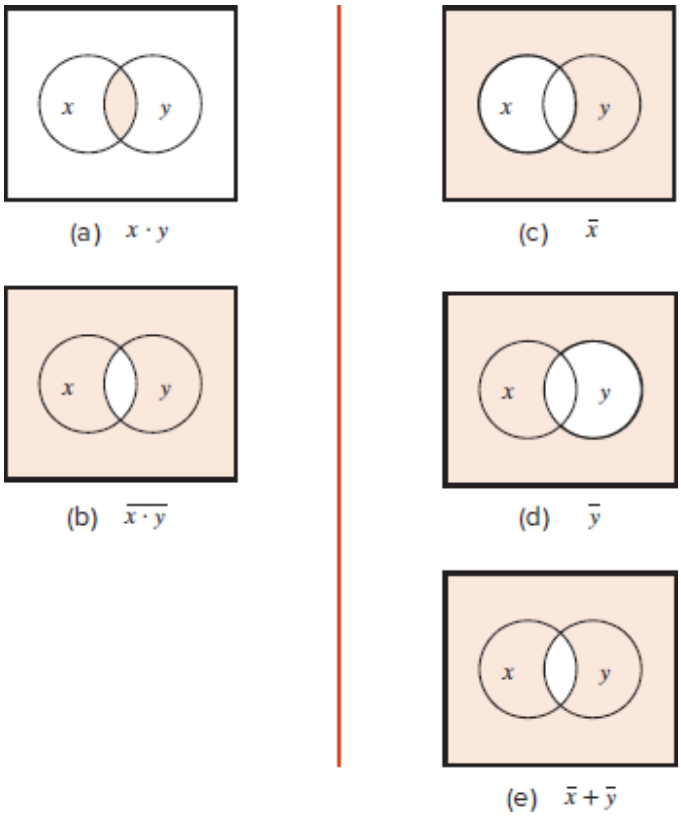


Figure 2.18 Proof of DeMorgan's theorem 15*a*.

2.5.2 NOTATION AND TERMINOLOGY

Boolean algebra is based on the AND and OR operations, for which we have adopted the symbols · and +, respectively. These are also the standard symbols for the familiar arithmetic multiplication and addition operations. Considerable similarity exists between the Boolean operations and the arithmetic operations, which is the main reason why the same symbols are used. In fact, when single digits are involved there is only one significant difference; the result of 1 + 1 is equal to 2 in ordinary arithmetic, whereas it is equal to 1 in Boolean algebra as defined by theorem 7*b* in Section 2.5.

Because of the similarity with the arithmetic addition and multiplication operations, the OR and AND operations are often called the *logical sum* and *product* operations. Thus $x_1 + x_2$ is the logical sum of x_1 and x_2, and $x_1 \cdot x_2$ is the logical product of x_1 and x_2. Instead of saying "logical product" and "logical sum," it is customary to say simply "product" and "sum." Thus we say that the expression

$$x_1 \cdot \bar{x}_2 \cdot x_3 + \bar{x}_1 \cdot x_4 + x_2 \cdot x_3 \cdot \bar{x}_4$$

is a sum of three product terms, whereas the expression

$$(\bar{x}_1 + x_3) \cdot (x_1 + \bar{x}_3) \cdot (\bar{x}_2 + x_3 + x_4)$$

is a product of three sum terms.

2.5.3 PRECEDENCE OF OPERATIONS

Using the three basic operations—AND, OR, and NOT—it is possible to construct an infinite number of logic expressions. Parentheses can be used to indicate the order in which the operations should be performed. However, to avoid an excessive use of parentheses, another convention defines the precedence of the basic operations. It states that in the absence of parentheses, operations in a logic expression must be performed in the order: NOT, AND, and then OR. Thus, in the expression

$$x_1 \cdot x_2 + \bar{x}_1 \cdot \bar{x}_2$$

it is first necessary to generate the complements of x_1 and x_2. Then the product terms $x_1 \cdot x_2$ and $\bar{x}_1 \cdot \bar{x}_2$ are formed, followed by the sum of the two product terms. Observe that in the absence of this convention, we would have to use parentheses to achieve the same effect as follows:

$$(x_1 \cdot x_2) + ((\bar{x}_1) \cdot (\bar{x}_2))$$

Finally, to simplify the appearance of logic expressions, it is customary to omit the $\cdot$ operator when there is no ambiguity. Therefore, the preceding expression can be written as

$$x_1 x_2 + \bar{x}_1 \bar{x}_2$$

We will use this style throughout the book.

2.6 Synthesis Using AND, OR, and NOT Gates

Armed with some basic ideas, we can now try to implement arbitrary functions using the AND, OR, and NOT gates. Suppose that we wish to design a logic circuit with two inputs, x_1 and x_2. Assume that x_1 and x_2 represent the states of two switches, either of which may produce a 0 or 1. The function of the circuit is to continuously monitor the state of the switches and to produce an output logic value 1 whenever the switches (x_1, x_2) are in states (0, 0), (0, 1), or (1, 1). If the state of the switches is (1, 0), the output should be 0. We can express the required behavior using a truth table, as shown in Figure 2.19.

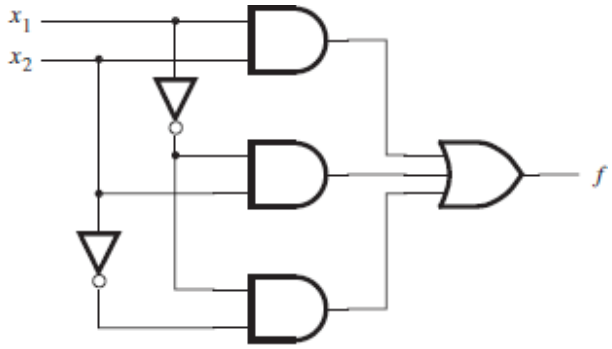
x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

Figure 2.19 A function to be synthesized.

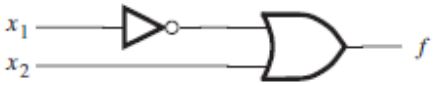
A possible procedure for designing a logic circuit that implements this truth table is to create a product term that has a value of 1 for each valuation for which the output function f has to be 1. Then we can take a logical sum of these product terms to realize f. Let us begin with the fourth row of the truth table, which corresponds to $x_1 = x_2 = 1$. The product term that is equal to 1 for this valuation is $x_1 \cdot x_2$, which is just the AND of x_1 and x_2. Next consider the first row of the table, for which $x_1 = x_2 = 0$. For this valuation the value 1 is produced by the product term $\bar{x}_1 \cdot \bar{x}_2$. Similarly, the second row leads to the term $\bar{x}_1 \cdot x_2$. Thus f may be realized as

$$f(x_1, x_2) = x_1 x_2 + \bar{x}_1 \bar{x}_2 + \bar{x}_1 x_2$$

The logic network that corresponds to this expression is shown in Figure 2.20*a*.



(a) Canonical sum-of-products



(b) Minimal-cost realization

Figure 2.20 Two implementations of the function in Figure 2.19.

Although this network implements f correctly, it is not the simplest such network. To find a simpler network, we can manipulate the obtained expression using the theorems and properties from Section 2.5. According to theorem 7*b*, we can replicate any term in a logical sum expression. Replicating the third product term, the above expression becomes

$$f(x_1, x_2) = x_1x_2 + \bar{x}_1\bar{x}_2 + \bar{x}_1x_2 + \bar{x}_1x_2$$

Using the commutative property 10*b* to interchange the second and third product terms gives

$$f(x_1, x_2) = x_1x_2 + \bar{x}_1x_2 + \bar{x}_1\bar{x}_2 + \bar{x}_1x_2$$

Now the distributive property 12*a* allows us to write

$$f(x_1, x_2) = (x_1 + \bar{x}_1)x_2 + \bar{x}_1(\bar{x}_2 + x_2)$$

Applying theorem 8*b* we get

$$f(x_1, x_2) = 1 \cdot x_2 + \bar{x}_1 \cdot 1$$

Finally, theorem 6*a* leads to

$$f(x_1, x_2) = x_2 + \bar{x}_1$$

The network described by this expression is given in Figure 2.20*b*. Obviously, the cost of this network is much less than the cost of the network in part (*a*) of the figure.

This simple example illustrates two things. First, a straightforward implementation of a function can be obtained by using a product term (AND gate) for each row of the truth table for which the function is equal to 1. Each product term contains all input variables, and it is formed such that if the input variable x_i is equal to 1 in the given row, then x_i is entered in the term; if $x_i = 0$ in that row, then $\bar{x}_i$ is entered. The sum of these product terms realizes the desired function. Second, there are many different networks that can realize a given function. Some of these networks may be simpler than others. Algebraic manipulation can be used to derive simplified logic expressions and thus lower-cost networks.

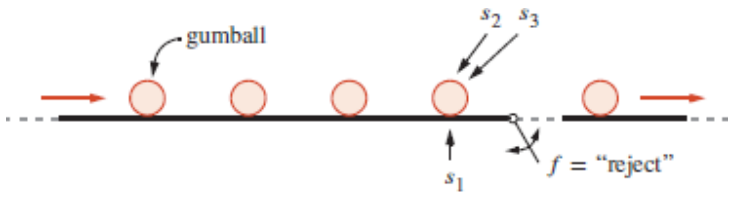
The process whereby we begin with a description of the desired functional behavior and then generate a circuit that realizes this behavior is called *synthesis*. Thus we can say that we "synthesized" the networks in Figure 2.20 from the truth table in Figure 2.19. Generation of AND-OR expressions from a truth table is just one of many types of synthesis techniques that we will encounter in this book.

Example 2.7

Figure 2.21*a* depicts a part of a factory that makes bubble gumballs. The gumballs travel on a conveyor that has three associated sensors s_1, s_2, and s_3. The sensor s_1 is connected to a scale that weighs each gumball, and if a gumball is not heavy enough to be acceptable then the sensor sets $s_1 = 1$. Sensors s_2 and s_3 examine the diameter of each gumball. If a gumball is too small to be acceptable, then $s_2 = 1$, and if it is too large, then $s_3 = 1$. If a gumball is of an acceptable weight and size, then the sensors give $s_1 = s_2 = s_3 = 0$. The conveyor pushes the gumballs over a "trap door" that it used to reject the ones that are not properly formed. A gumball should be rejected if it is too large, or both too small and too light. The trap door is opened by setting the logic function f to the value 1. By inspection, we can see that an appropriate

logic expression is $f = s_1s_2 + s_3$. We will use Boolean algebra to derive this logic expression from the truth table.

The truth table for f is given in Figure 2.21*b*. It sets f to 1 for each row in the table where s_3 has the value 1 (too large), as well as for each row where $s_1 = s_2 = 1$ (too light and too small). As described previously, a logic expression for f can be formed by including a product term for each row where f=1. Thus, we can write



(a) Conveyor and sensors

s_1	s_2	s_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(b) Truth table

Figure 2.21 A bubble gumball factory.

$$f = \bar{s}_1\bar{s}_2s_3 + \bar{s}_1s_2s_3 + s_1\bar{s}_2s_3 + s_1s_2\bar{s}_3 + s_1s_2s_3$$

We can use algebraic manipulation to simplify this expression in a number of ways. For example, as shown below, we can first use rule 7*b* to repeat the term $s_1s_2s_3$, and then use the distributive property 12*a* and rule 8*b* to simplify the expression

$$\begin{aligned} f &= \bar{s}_1\bar{s}_2s_3 + \bar{s}_1s_2s_3 + s_1\bar{s}_2s_3 + s_1s_2s_3 + s_1s_2\bar{s}_3 + s_1s_2s_3 \\ &= \bar{s}_1s_3(\bar{s}_2 + s_2) + s_1s_3(\bar{s}_2 + s_2) + s_1s_2(\bar{s}_3 + s_3) \\ &= \bar{s}_1s_3 + s_1s_3 + s_1s_2 \end{aligned}$$

Now, using the combining property 14*a* on the first two product terms gives

$$f = s_3 + s_1 s_2$$

The observant reader will notice that using the combining property 14*a* is really just a short form of first using the distributive property 12*a* and then applying rule 8*b*, as we did in the previous step. Our simplified expression for *f* is the same as the one that we determined earlier, by inspection.

Example 2.8

There are different ways in which we can simplify the logic expression produced from the truth table in Figure 2.21*b*. Another approach is to first repeat the term $s_1 s_2 s_3$, as we did in Example 2.7, and then proceed as follows:

$$\begin{aligned} f &= \bar{s}_1 \bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1 \bar{s}_2 s_3 + s_1 s_2 s_3 + s_1 s_2 \bar{s}_3 + s_1 s_2 s_3 \\ &= s_3(\bar{s}_1 \bar{s}_2 + \bar{s}_1 s_2 + s_1 \bar{s}_2 + s_1 s_2) + s_1 s_2(\bar{s}_3 + s_3) \\ &= s_3 \cdot 1 + s_1 s_2 \\ &= s_3 + s_1 s_2 \end{aligned}$$

Here, we used the distributive property 12*a* to produce the expression $(\bar{s}_1 \bar{s}_2 + \bar{s}_1 s_2 + s_1 \bar{s}_2 + s_1 s_2)$. Since this expression includes all possible valuations of s_1, s_2, it is equal to 1, leading to the same expression for *f* that we derived before.

Example 2.9

Yet another way of producing the simplified logic expression is shown below.

$$\begin{aligned} f &= \bar{s}_1 \bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1 \bar{s}_2 s_3 + s_1 s_2 \bar{s}_3 + s_1 s_2 s_3 \\ &= \bar{s}_1 \bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1 \bar{s}_2 s_3 + \bar{s}_1 \bar{s}_2 s_3 + s_1 s_2 \bar{s}_3 + s_1 s_2 s_3 \\ &= \bar{s}_1 s_3(\bar{s}_2 + s_2) + \bar{s}_2 s_3(s_1 + \bar{s}_1) + s_1 s_2(\bar{s}_3 + s_3) \\ &= \bar{s}_1 s_3 + \bar{s}_2 s_3 + s_1 s_2 \\ &= s_3(\bar{s}_1 + \bar{s}_2) + s_1 s_2 \\ &= s_3(\overline{s_1 s_2}) + s_1 s_2 \\ &= s_3 + s_1 s_2 \end{aligned}$$

In this solution, we first repeat the term $\bar{s}_1 \bar{s}_2 s_3$ and then simplify to generate the expression $s_3(\bar{s}_1 + \bar{s}_2) + s_1 s_2$. Using DeMorgan's theorem 15*a*, we can replace $(\bar{s}_1 + \bar{s}_2)$ with $(\overline{s_1 s_2})$, which can then be deleted by applying property 16*a*.

As illustrated by Examples 2.7 to 2.9, there are multiple ways in which a logic expression can be minimized by using Boolean algebra. This process can be daunting, because it is not obvious which rules, identities, and properties should be applied, and in what order. Later in this chapter, in Section 2.11, we will introduce a graphical technique, called the Karnaugh map, that clarifies this process by providing a systematic way of generating a minimal-cost logic expression for a function.

2.6.1 SUM-OF-PRODUCTS AND PRODUCT-OF-SUMS FORMS

Having introduced the synthesis process by means of simple examples, we will now present it in more formal terms using the terminology that is encountered in the technical literature. We will also show how the principle of duality, which was introduced in Section 2.5, applies broadly in the synthesis process.

If a function f is specified in the form of a truth table, then an expression that realizes f can be obtained either by considering the rows in the table for which $f = 1$, as we have already done, or by considering the rows for which $f = 0$, as we will explain shortly.

Minterms

For a function of n variables, a product term in which each of the n variables appears once is called a *minterm*. The variables may appear in a minterm in either uncomplemented or complemented form. For a given row of the truth table, the minterm is formed by including x_i if $x_i = 1$ and by including $\bar{x}_i$ if $x_i = 0$.

To illustrate this concept, consider the truth table in Figure 2.22. We have numbered the rows of the table from 0 to 7, so that we can refer to them easily. From the discussion of the binary number representation in Section 1.5, we can observe that the row numbers chosen are just the numbers represented by the bit patterns of variables x_1, x_2, and x_3. The figure shows all minterms for the three-variable table. For example, in the first row the variables have the values $x_1 = x_2 = x_3 = 0$, which leads to the minterm $\bar{x}_1\bar{x}_2\bar{x}_3$. In the second row $x_1 = x_2 = 0$ and $x_3 = 1$, which gives the minterm $\bar{x}_1\bar{x}_2x_3$, and so on. To be able to refer to the individual minterms easily, it is convenient to identify each minterm by an index that corresponds to the row numbers

shown in the figure. We will use the notation m_i to denote the minterm for row number i. Thus, $m_0 = \bar{x}_1\bar{x}_2\bar{x}_3$, $m_1 = \bar{x}_1\bar{x}_2x_3$, and so on.

Row number	x_1	x_2	x_3	Minterm	Maxterm
0	0	0	0	$m_0 = \bar{x}_1\bar{x}_2\bar{x}_3$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \bar{x}_1\bar{x}_2x_3$	$M_1 = x_1 + x_2 + \bar{x}_3$
2	0	1	0	$m_2 = \bar{x}_1x_2\bar{x}_3$	$M_2 = x_1 + \bar{x}_2 + x_3$
3	0	1	1	$m_3 = \bar{x}_1x_2x_3$	$M_3 = x_1 + \bar{x}_2 + \bar{x}_3$
4	1	0	0	$m_4 = x_1\bar{x}_2\bar{x}_3$	$M_4 = \bar{x}_1 + x_2 + x_3$
5	1	0	1	$m_5 = x_1\bar{x}_2x_3$	$M_5 = \bar{x}_1 + x_2 + \bar{x}_3$
6	1	1	0	$m_6 = x_1x_2\bar{x}_3$	$M_6 = \bar{x}_1 + \bar{x}_2 + x_3$
7	1	1	1	$m_7 = x_1x_2x_3$	$M_7 = \bar{x}_1 + \bar{x}_2 + \bar{x}_3$

Figure 2.22 Three-variable minterms and maxterms.

Sum-of-Products Form

A function f can be represented by an expression that is a sum of minterms, where each minterm is ANDed with the value of f for the corresponding valuation of input variables. For example, the two-variable minterms are $m_0 = \bar{x}_1\bar{x}_2$, $m_1 = \bar{x}_1x_2$, $m_2 = x_1\bar{x}_2$, and $m_3 = x_1\ x_2$. The function in Figure 2.19 can be represented as

$$\begin{aligned} f &= m_0 \cdot 1 + m_1 \cdot 1 + m_2 \cdot 0 + m_3 \cdot 1 \\ &= m_0 + m_1 + m_3 \\ &= \bar{x}_1\bar{x}_2 + \bar{x}_1x_2 + x_1x_2 \end{aligned}$$

which is the form that we derived in the previous section using an intuitive approach. Only the minterms that correspond to the rows for which $f = 1$ appear in the resulting expression.

Any function f can be represented by a sum of minterms that correspond to the rows in the truth table for which $f = 1$. The resulting implementation is functionally correct and unique, but it is not necessarily the lowest-cost implementation of f. A logic expression consisting of product (AND) terms that are summed (ORed) is said to be in the *sum-of-products* (*SOP*) form. If each product term is a minterm, then the expression is called a *canonical sum-of-products* for the function f. As we have seen in the example of Figure 2.20, the first step in the synthesis process is to derive a canonical sum-of-products expression for the given function. Then we can manipulate this expression,

using the theorems and properties of Section 2.5, with the goal of finding a functionally equivalent sum-of-products expression that has a lower cost.

As another example, consider the three-variable function $f(x_1, x_2, x_3)$, specified by the truth table in Figure 2.23. To synthesize this function, we have to include the minterms m_1, m_4, m_5, and m_6. Copying these minterms from Figure 2.22 leads to the following canonical sum-of-products expression for *f*:

Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

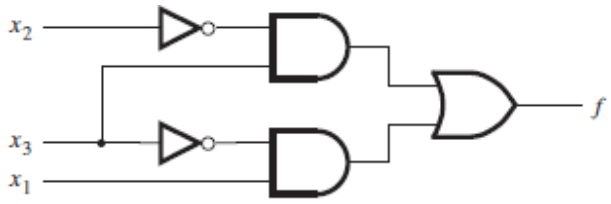
Figure 2.23 A three-variable function.

$$f(x_1, x_2, x_3) = \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3$$

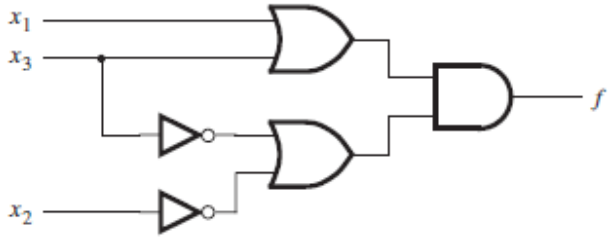
This expression can be manipulated as follows:

$$\begin{aligned} f &= (\bar{x}_1 + x_1)\bar{x}_2x_3 + x_1(\bar{x}_2 + x_2)\bar{x}_3 \\ &= 1 \cdot \bar{x}_2x_3 + x_1 \cdot 1 \cdot \bar{x}_3 \\ &= \bar{x}_2x_3 + x_1\bar{x}_3 \end{aligned}$$

This is the minimum-cost sum-of-products expression for *f*. It describes the circuit shown in Figure 2.24*a*. A good indication of the *cost* of a logic circuit is the total number of gates plus the total number of inputs to all gates in the circuit. Using this measure, the cost of the network in Figure 2.24*a* is 13, because there are five gates and eight inputs to the gates. By comparison, the network implemented on the basis of the canonical sum-of-products would have a cost of 27; from the preceding expression, the OR gate has four inputs, each of the four AND gates has three inputs, and each of the three NOT gates has one input.



(a) A minimal sum-of-products realization



(b) A minimal product-of-sums realization

Figure 2.24 Two realizations of the function in Figure 2.23.

Minterms, with their row-number subscripts, can also be used to specify a given function in a more concise form. For example, the function in Figure 2.23 can be specified as

$$f(x_1, x_2, x_3) = \sum(m_1, m_4, m_5, m_6)$$

or even more simply as

$$f(x_1, x_2, x_3) = \sum m(1, 4, 5, 6)$$

The $\sum$ sign denotes the logical sum operation. This shorthand notation is often used in practice.

Example 2.10

consider the function

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

The canonical SOP expression for the function is derived using minterms

$$f = m_2 + m_3 + m_4 + m_6 + m_7$$
$$= \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3$$

This expression can be simplified using the identities in Section 2.5 as follows:

$$f = \bar{x}_1 x_2 (\bar{x}_3 + x_3) + x_1 (\bar{x}_2 + x_2) \bar{x}_3 + x_1 x_2 (\bar{x}_3 + x_3)$$
$$= \bar{x}_1 x_2 + x_1 \bar{x}_3 + x_1 x_2$$
$$= (\bar{x}_1 + x_1) x_2 + x_1 \bar{x}_3$$
$$= x_2 + x_1 \bar{x}_3$$

Example 2.11

suppose that a four-variable function is defined by

$$f(x_1, x_2, x_3, x_4) = \sum m(3, 7, 9, 12, 13, 14, 15)$$

The canonical SOP expression for this function is

$$f = \bar{x}_1 \bar{x}_2 x_3 x_4 + \bar{x}_1 x_2 x_3 x_4 + x_1 \bar{x}_2 \bar{x}_3 x_4 + x_1 x_2 \bar{x}_3 \bar{x}_4 + x_1 x_2 \bar{x}_3 x_4 + x_1 x_2 x_3 \bar{x}_4 + x_1 x_2 x_3 x_4$$

A simpler SOP expression can be obtained as follows:

$$f = \bar{x}_1 (\bar{x}_2 + x_2) x_3 x_4 + x_1 (\bar{x}_2 + x_2) \bar{x}_3 x_4 + x_1 x_2 \bar{x}_3 (\bar{x}_4 + x_4) + x_1 x_2 x_3 (\bar{x}_4 + x_4)$$
$$= \bar{x}_1 x_3 x_4 + x_1 \bar{x}_3 x_4 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3$$
$$= \bar{x}_1 x_3 x_4 + x_1 \bar{x}_3 x_4 + x_1 x_2 (\bar{x}_3 + x_3)$$
$$= \bar{x}_1 x_3 x_4 + x_1 \bar{x}_3 x_4 + x_1 x_2$$

Maxterms

The principle of duality suggests that if it is possible to synthesize a function f by considering the rows in the truth table for which $f = 1$, then it should also be possible to synthesize f by considering the rows for which $f = 0$. This alternative approach uses the complements of minterms, which are called *maxterms*. All possible maxterms for three-variable functions are listed in Figure 2.22. We will refer to a maxterm M_j by the same row number as its corresponding minterm m_j as shown in the figure.

Product-of-Sums Form

If a given function f is specified by a truth table, then its complement $\bar{f}$ can be represented by a sum of minterms for which $\bar{f} = 1$, which are the rows where $f = 0$. For example, for the function in Figure 2.19

$$\begin{aligned}\bar{f}(x_1, x_2) &= m_2 \\ &= x_1\bar{x}_2\end{aligned}$$

If we complement this expression using DeMorgan's theorem, the result is

$$\begin{aligned}\bar{\bar{f}} = f &= \overline{x_1\bar{x}_2} \\ &= \bar{x}_1 + x_2\end{aligned}$$

Note that we obtained this expression previously by algebraic manipulation of the canonical sum-of-products form for the function f. The key point here is that

$$f = \bar{m}_2 = M_2$$

where M_2 is the maxterm for row 2 in the truth table.

As another example, consider again the function in Figure 2.23. The complement of this function can be represented as

$$\begin{aligned}\bar{f}(x_1, x_2, x_3) &= m_0 + m_2 + m_3 + m_7 \\ &= \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_1x_2x_3 + x_1x_2x_3\end{aligned}$$

Then f can be expressed as

$$\begin{aligned}f &= \overline{m_0 + m_2 + m_3 + m_7} \\ &= \bar{m}_0 \cdot \bar{m}_2 \cdot \bar{m}_3 \cdot \bar{m}_7 \\ &= M_0 \cdot M_2 \cdot M_3 \cdot M_7 \\ &= (x_1 + x_2 + x_3)(x_1 + \bar{x}_2 + x_3)(x_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)\end{aligned}$$

This expression represents f as a product of maxterms.

A logic expression consisting of sum (OR) terms that are the factors of a logical product (AND) is said to be of the *product-of-sums* (*POS*) form. If each sum term is a maxterm, then the expression is called a *canonical product-of-sums* for the given function. Any function f can be synthesized by finding its canonical product-of-sums. This involves taking the maxterm for

each row in the truth table for which $f = 0$ and forming a product of these maxterms.

Returning to the preceding example, we can attempt to reduce the complexity of the derived expression that comprises a product of maxterms. Using the commutative property 10*b* and the associative property 11*b* from Section 2.5, this expression can be written as

$$f = ((x_1 + x_3) + x_2)((x_1 + x_3) + \bar{x}_2)(x_1 + (\bar{x}_2 + \bar{x}_3))(\bar{x}_1 + (\bar{x}_2 + \bar{x}_3))$$

Then, using the combining property 14*b*, the expression reduces to

$$f = (x_1 + x_3)(\bar{x}_2 + \bar{x}_3)$$

The corresponding network is given in Figure 2.24*b*. The cost of this network is 13. While this cost happens to be the same as the cost of the sum-of-products version in Figure 2.24*a*, the reader should not assume that the cost of a network derived in the sum-of-products form will in general be equal to the cost of a corresponding circuit derived in the product-of-sums form.

Using the shorthand notation, an alternative way of specifying our sample function is

$$f(x_1, x_2, x_3) = \Pi(M_0, M_2, M_3, M_7)$$

or more simply

$$f(x_1, x_2, x_3) = \Pi M(0, 2, 3, 7)$$

The Π sign denotes the logical product operation.

The preceding discussion has shown how logic functions can be realized in the form of logic circuits, consisting of networks of gates that implement basic functions. A given function may be realized with various different circuit structures, which usually implies a difference in cost. An important objective for a designer is to minimize the cost of the designed circuit. We will discuss strategies for finding minimum-cost implementations in Section 2.11.

Example 2.12

Consider again the function in Example 2.10. Instead of using the minterms, we can specify this function as a product of maxterms for which $f = 0$, namely

$$f(x_1, x_2, x_3) = \Pi M(0, 1, 5)$$

Then, the canonical POS expression is derived as

$$\begin{aligned} f &= M_0 \cdot M_1 \cdot M_5 \\ &= (x_1 + x_2 + x_3)(x_1 + x_2 + \bar{x}_3)(\bar{x}_1 + x_2 + \bar{x}_3) \end{aligned}$$

A simplified POS expression can be derived as

$$\begin{aligned} f &= (x_1 + x_2 + x_3)(x_1 + x_2 + \bar{x}_3)(x_1 + x_2 + \bar{x}_3)(\bar{x}_1 + x_2 + \bar{x}_3) \\ &= ((x_1 + x_2) + x_3)((x_1 + x_2) + \bar{x}_3)(x_1 + (x_2 + \bar{x}_3))(\bar{x}_1 + (x_2 + \bar{x}_3)) \\ &= ((x_1 + x_2) + x_3\bar{x}_3)(x_1\bar{x}_1 + (x_2 + \bar{x}_3)) \\ &= (x_1 + x_2)(x_2 + \bar{x}_3) \end{aligned}$$

Another way of deriving this product-of-sums expression is to use the sum-of-products form of $\bar{f}$. Thus,

$$\begin{aligned} \bar{f}(x_1, x_2, x_3) &= \sum m(0, 1, 5) \\ &= \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2x_3 \\ &= \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2x_3 \\ &= \bar{x}_1\bar{x}_2(\bar{x}_3 + x_3) + \bar{x}_2x_3(\bar{x}_1 + x_1) \\ &= \bar{x}_1\bar{x}_2 + \bar{x}_2x_3 \end{aligned}$$

Now, first applying DeMorgan's theorem 15*b*, and then applying 15*a* (twice) gives

$$\begin{aligned} f &= \bar{\bar{f}} \\ &= \overline{(\bar{x}_1\bar{x}_2 + \bar{x}_2x_3)} \\ &= \overline{(\bar{x}_1\bar{x}_2)}\,\overline{(\bar{x}_2x_3)} \\ &= (x_1 + x_2)(x_2 + \bar{x}_3) \end{aligned}$$

To see that this product-of-sums expression for f is equivalent to the sum-of-products expression that we derived in Example 2.10, we can slightly rearrange our expression as $f = (x_2 + x_1)(x_2 + \bar{x}_3)$. Now, recognizing that this expression has the form of the right-hand side of the distributive property 12*b*, we have the sum-of-products expression $f = x_2 + x_1\bar{x}_3$.

2.7 NAND AND NOR LOGIC NETWORKS

We have discussed the use of AND, OR, and NOT gates in the synthesis of logic circuits. There are other basic logic functions, that are also used for this purpose. Particularly useful are the NAND and NOR functions, which are obtained by complementing the output generated by AND and OR operations, respectively. These functions are attractive because they are implemented with simpler electronic circuits than the AND and OR functions, as we discuss in Appendix B. Figure 2.25 gives the graphical symbols for the NAND and NOR gates. A bubble is placed on the output side of the AND and OR gate symbols to represent the complemented output signal.

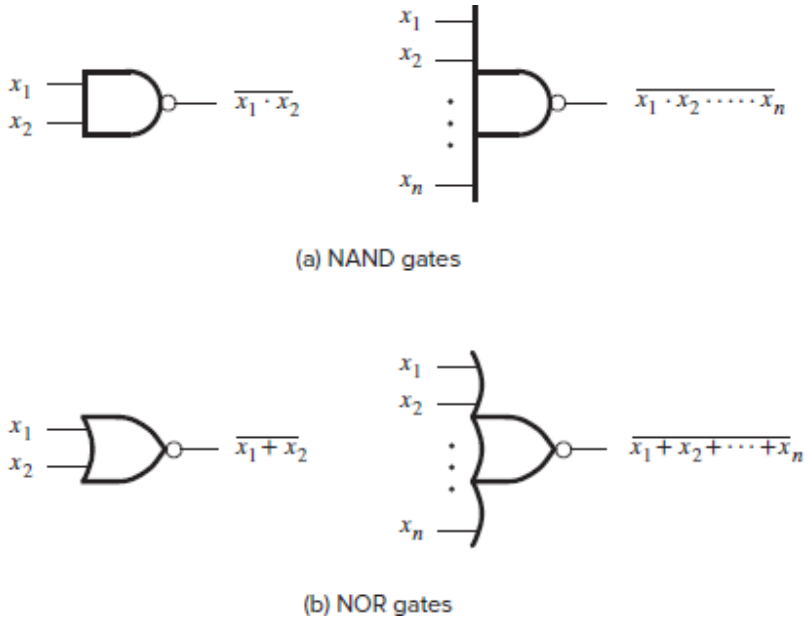


Figure 2.25 NAND and NOR gates.

If NAND and NOR gates are realized with simpler circuits than AND and OR gates, then we should ask whether these gates can be used directly in the synthesis of logic circuits. In Section 2.5 we introduced DeMorgan's theorem. Its logic gate interpretation is shown in Figure 2.26. Identity 15*a* is interpreted in part (*a*) of the figure. It specifies that a NAND of variables x_1 and x_2 is equivalent to first complementing each of the variables and then ORing them. Notice on the far-right side that we have indicated the NOT gates simply as bubbles, which denote inversion of the logic value at that point. The other half of DeMorgan's theorem, identity 15*b*, appears in part (*b*) of the figure. It states

that the NOR function is equivalent to first inverting the input variables and then ANDing them.

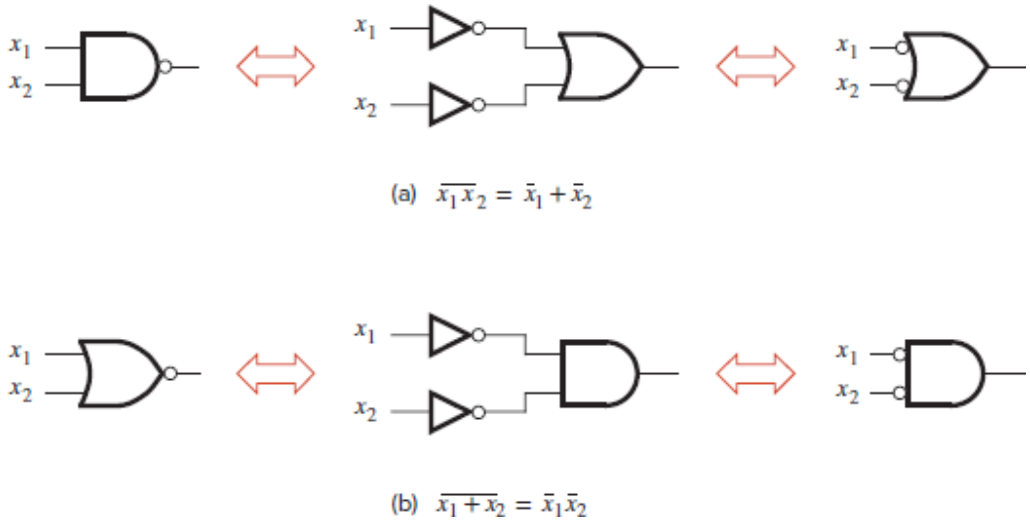


Figure 2.26 DeMorgan's theorem in terms of logic gates.

In Section 2.6 we explained how any logic function can be implemented in either sum-of-products or product-of-sums form, which leads to logic networks that have either an AND-OR or an OR-AND structure, respectively. We will now show that such networks can be implemented using only NAND gates or only NOR gates.

Consider the network in Figure 2.27 as a representative of general AND-OR networks. This network can be transformed into a network of NAND gates as shown in the figure. First, each connection between the AND gate and an OR gate is replaced by a connection that includes two inversions of the signal: one inversion at the output of the AND gate and the other at the input of the OR gate. Such double inversion has no effect on the behavior of the network, as stated formally in theorem 9 in Section 2.5. According to Figure 2.26*a*, the OR gate with inversions at its inputs is equivalent to a NAND gate. Thus we can redraw the network using only NAND gates, as shown in Figure 2.27. This example shows that any AND-OR network can be implemented as a NAND-NAND network having the same topology.

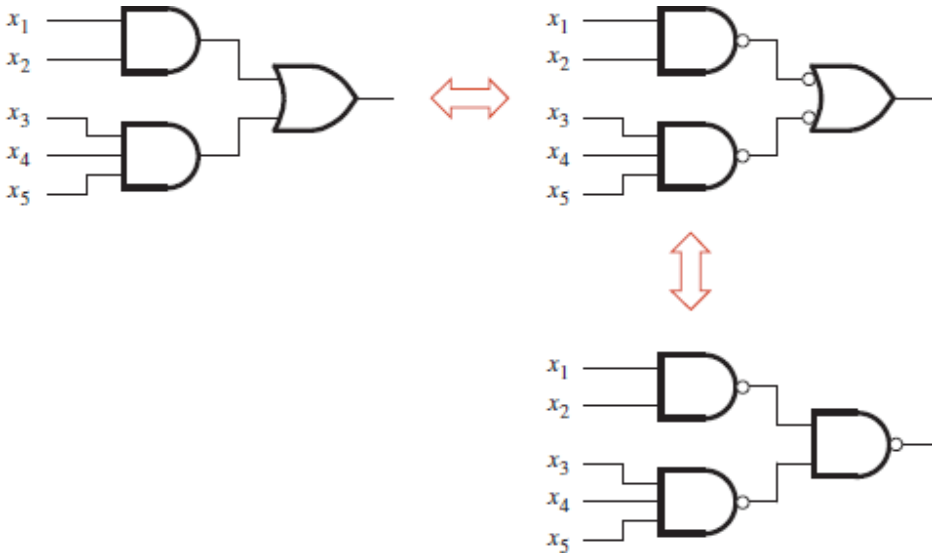


Figure 2.27 Using NAND gates to implement a sum-of-products.

Figure 2.28 gives a similar construction for a product-of-sums network, which can be transformed into a circuit with only NOR gates. The procedure is exactly the same as the one described for Figure 2.27 except that now the identity in Figure 2.26*b* is applied. The conclusion is that any OR-AND network can be implemented as a NOR-NOR network having the same topology.

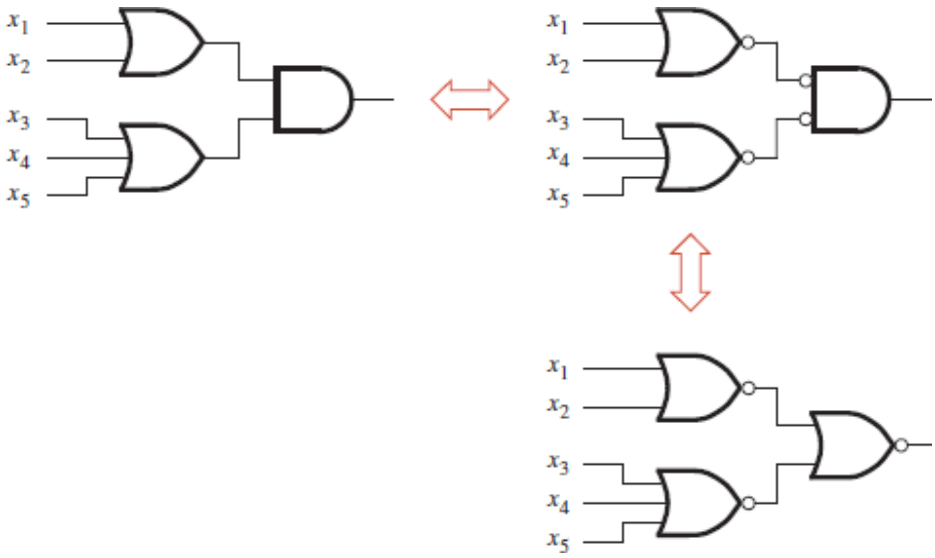


Figure 2.28 Using NOR gates to implement a product-of-sums.

Example 2.13

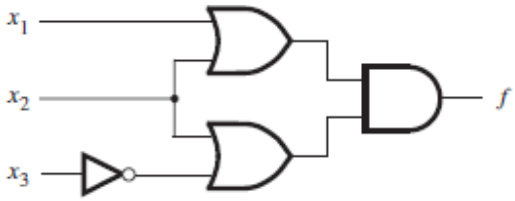
Let us implement the function

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

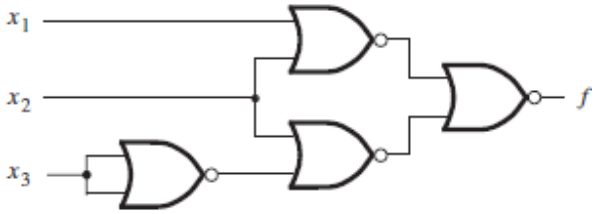
using NOR gates only. In Example 2.12 we showed that the function can be represented by the POS expression

$$f = (x_1 + x_2)(x_2 + \bar{x}_3)$$

An OR-AND circuit that corresponds to this expression is shown in Figure 2.29*a*. Using the same structure of the circuit, a NOR-gate version is given in Figure 2.29*b*. Note that x_3 is inverted by a NOR gate that has its inputs tied together.



(a) POS implementation



(b) NOR implementation

Figure 2.29 NOR-gate realization of the function in Example 2.13.

Example 2.14

Let us now implement the function

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

using NAND gates only. In Example 2.10 we derived the SOP expression

$$f = x_2 + x_1\bar{x}_3$$

which is realized using the circuit in Figure 2.30*a*. We can again use the same structure to obtain a circuit with NAND gates, but with one difference. The signal x_2 passes only through an OR gate, instead of passing through an AND gate and an OR gate. If we simply replace the OR gate with a NAND gate, this signal would be inverted, which would result in a wrong output value. Since x_2 must either not be inverted, or it can be inverted twice, we can pass it through two NAND gates as depicted in Figure 2.30*b*. Observe that for this circuit the output f is

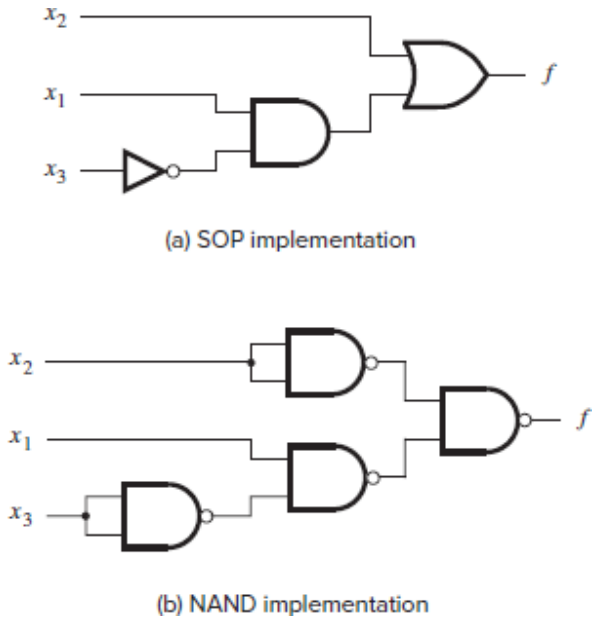


Figure 2.30 NAND-gate realization of the function in Example 2.10.

$$f = \overline{\bar{x}_2 \cdot \overline{x_1\bar{x}_3}}$$

Applying DeMorgan's theorem, this expression becomes

$$f = x_2 + x_1\bar{x}_3$$

2.8 Design Examples

Logic circuits provide a solution to a problem. They implement functions that are needed to carry out specific tasks. Within the framework of a computer, logic circuits provide complete capability for execution of programs and processing of data. Such circuits are complex and difficult to design. But regardless of the complexity of a given circuit, a designer of logic circuits is always confronted with the same basic issues. First, it is necessary to specify the desired behavior of the circuit. Second, the circuit has to be synthesized and implemented. Finally, the implemented circuit has to be tested to verify that it meets the specifications. The desired behavior is often initially described in words, which then must be turned into a formal specification. In this section we give three simple examples of design.

2.8.1 THREE-WAY LIGHT CONTROL

Assume that a large room has three doors and that a switch near each door controls a light in the room. It has to be possible to turn the light on or off by changing the state of any one of the switches.

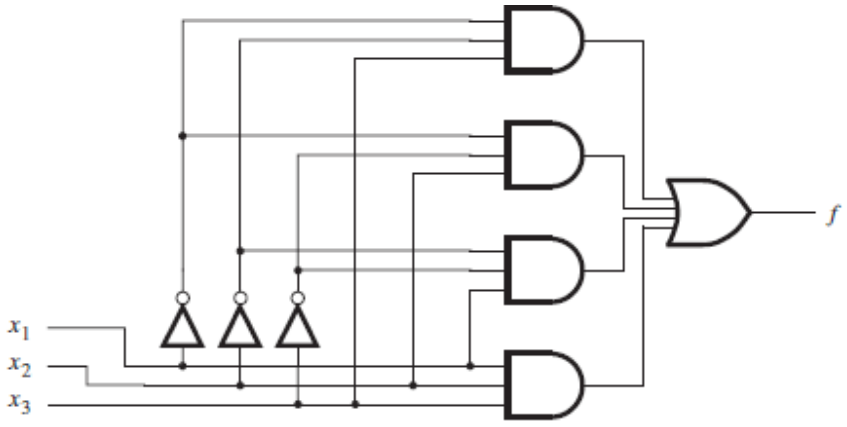
As a first step, let us turn this word statement into a formal specification using a truth table. Let x_1, x_2, and x_3 be the input variables that denote the state of each switch. Assume that the light is off if all switches are open. Closing any one of the switches will turn the light on. Then turning on a second switch will have to turn off the light. Thus the light will be on if exactly one switch is closed, and it will be off if two (or no) switches are closed. If the light is off when two switches are closed, then it must be possible to turn it on by closing the third switch. If $f(x_1, x_2, x_3)$ represents the state of the light, then the required functional behavior can be specified as shown in the truth table in Figure 2.31. The canonical sum-of-products expression for the specified function is

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

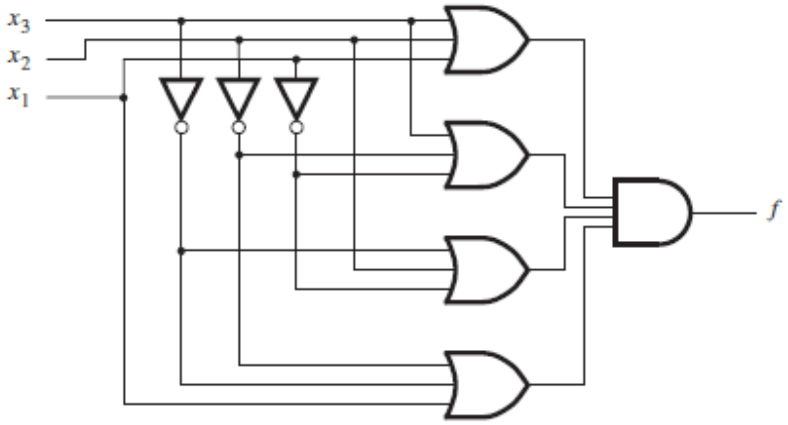
Figure 2.31 Truth table for the three-way light control.

$$f = m_1 + m_2 + m_4 + m_7$$
$$= \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_1x_2x_3$$

This expression cannot be simplified into a lower-cost sum-of-products expression. The resulting circuit is shown in Figure 2.32*a*.



(a) Sum-of-products realization



(b) Product-of-sums realization

Figure 2.32 Implementation of the function in Figure 2.31.

An alternative realization for this function is in the product-of-sums form. The canonical expression of this type is

$$f = M_0 \cdot M_3 \cdot M_5 \cdot M_6$$
$$= (x_1 + x_2 + x_3)(x_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + x_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_2 + x_3)$$

The resulting circuit is depicted in Figure 2.32*b*. It has the same cost as the circuit in part (*a*) of the figure.

When the designed circuit is implemented, it can be tested by applying the various input valuations to the circuit and checking whether the output corresponds to the values specified in the truth table. A straightforward approach is to check that the correct output is produced for all eight possible input valuations.

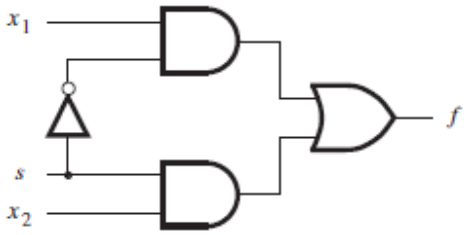
2.8.2 MULTIPLEXER CIRCUIT

In computer systems it is often necessary to choose data from exactly one of a number of possible sources. Suppose that there are two sources of data, provided as input signals x_1 and x_2. The values of these signals change in time, perhaps at regular intervals. Thus sequences of 0s and 1s are applied on each of the inputs x_1 and x_2. We want to design a circuit that produces an output that has the same value as either x_1 or x_2, dependent on the value of a selection control signal s. Therefore, the circuit should have three inputs: x_1, x_2, and s. Assume that the output of the circuit will be the same as the value of input x_1 if $s = 0$, and it will be the same as x_2 if $s = 1$.

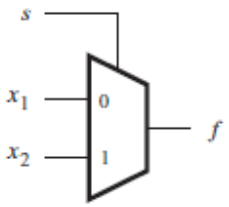
Based on these requirements, we can specify the desired circuit in the form of a truth table given in Figure 2.33*a*. From the truth table, we derive the canonical sum of products

$s\ x_1\ x_2$	$f(s, x_1, x_2)$
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1

(a) Truth table



(b) Circuit



(c) Graphical symbol

s	$f(s, x_1, x_2)$
0	x_1
1	x_2

(d) More compact truth-table representation

Figure 2.33 Implementation of a multiplexer.

$$f(s, x_1, x_2) = \bar{s}x_1\bar{x}_2 + \bar{s}x_1x_2 + s\bar{x}_1x_2 + sx_1x_2$$

Using the distributive property, this expression can be written as

$$f = \bar{s}x_1(\bar{x}_2 + x_2) + s(\bar{x}_1 + x_1)x_2$$

Applying theorem 8*b* yields

$$f = \bar{s}x_1 \cdot 1 + s \cdot 1 \cdot x_2$$

Finally, theorem 6*a* gives

$$f = \bar{s}x_1 + sx_2$$

A circuit that implements this function is shown in Figure 2.33*b*. Circuits of this type are used so extensively that they are given a special name. A circuit that generates an output that exactly reflects the state of one of a number of data inputs, based on the value of one or more selection control inputs, is called a *multiplexer*. We say that a multiplexer circuit "multiplexes" input signals onto a single output.

In this example we derived a multiplexer with two data inputs, which is referred to as a "2-to-1 multiplexer." A commonly used graphical symbol for the 2-to-1 multiplexer is shown in Figure 2.33*c*. The same idea can be extended to larger circuits. A 4-to-1 multiplexer has four data inputs and one output. In this case two selection control inputs are needed to choose one of the four data inputs that is transmitted as the output signal. An 8-to-1 multiplexer needs eight data inputs and three selection control inputs, and so on.

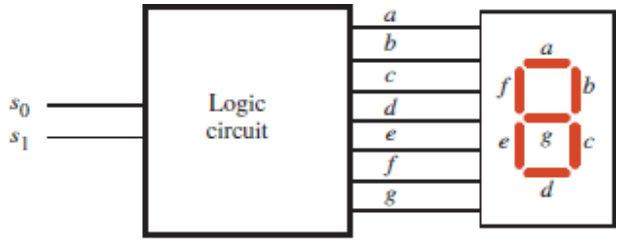
Note that the statement "$f = x_1$ if $s = 0$, and $f = x_2$ if $s = 1$" can be presented in a more compact form of a truth table, as indicated in Figure 2.33*d*. In later chapters we will have occasion to use such representation.

We showed how a multiplexer can be built using AND, OR, and NOT gates. The same circuit structure can be used to implement the multiplexer using NAND gates, as explained in Section 2.7. In Appendix B we will show other possibilities for constructing multiplexers. In Chapter 4 we will discuss the use of multiplexers in considerable detail.

2.8.3 NUMBER DISPLAY

In Example 2.2 we designed an adder circuit that generates the arithmetic sum $S = a + b$, where a and b are one-bit numbers and $S = s_1 s_0$ provides the resulting two-bit sum, which is either 00, 01, or 10. In this design example, we wish to create a logic circuit that drives a familiar seven-segment display, as illustrated in Figure 2.34*a*. This display allows us to show the value of S as a decimal number, either 0, 1, or 2. The display includes seven segments, labeled a, b, …, g in the figure, where each segment is a light-emitting diode (LED). Our logic circuit has the two inputs s_1 and s_0. It produces seven outputs, one for each segment in the display. Setting an output to the value 1

illuminates the corresponding segment in the display. By illuminating specific segments for each valuation of s_1 s_0 we can make the display's appearance correspond to the shape of the appropriate decimal digit.



(a) Logic circuit and 7-segment display

	s_1	s_0	a	b	c	d	e	f	g
0	0	0	1	1	1	1	1	1	0
1	0	1	0	1	1	0	0	0	0
2	1	0	1	1	0	1	1	0	1

(b) Truth table

Figure 2.34 Display of numbers.

Part (*b*) of Figure 2.34 shows a truth table for the possible valuations of s_1 s_0 and indicates on the left-hand side how the display should appear in each case. The truth table specifies the logic values needed for each of the seven functions. For example, segment *a* in the 7-segment display needs to be turned on when *S* has the decimal values 0 or 2, but has to be off when *S* has the value 1. Hence, the corresponding logic function is set to 1 for minterms m_0 and m_2, giving $a = \bar{s}_1\bar{s}_0 + s_1\bar{s}_0 = \bar{s}_0$. Logic expressions for each of the seven functions are

$$a = d = e = \bar{s}_0$$
$$b = 1$$
$$c = \bar{s}_1$$
$$f = \bar{s}_1\bar{s}_0$$
$$g = s_1\bar{s}_0$$

Designers of logic circuits rely heavily on CAD tools. We want to encourage the reader to become familiar with CAD tools as soon as possible. We have reached a point where an introduction to these tools is useful. The

next section presents some basic concepts that are needed to use these tools. We will also introduce, in Section 2.10, a special language for describing logic circuits, called VHDL. This language is used to describe the circuits as an input to the CAD tools, which then proceed to derive a suitable implementation.

2.9 Introduction to CAD Tools

The preceding sections introduced a basic approach for synthesis of logic circuits. A designer could use this approach manually for small circuits. However, logic circuits found in complex systems, such as today's computers, cannot be designed manually—they are designed using sophisticated CAD tools that automatically implement the synthesis techniques.

To design a logic circuit, a number of CAD tools are needed. They are usually packaged together into a *CAD system*, which typically includes tools for the following tasks: design entry, logic synthesis and optimization, simulation, and physical design. We will introduce some of these tools in this section and will provide additional discussion in later chapters.

2.9.1 Design Entry

The starting point in the process of designing a logic circuit is the conception of what the circuit is supposed to do and the formulation of its general structure. This step is done manually by the designer because it requires design experience and intuition. The rest of the design process is done with the aid of CAD tools. The first stage of this process involves entering into the CAD system a description of the circuit being designed. This stage is called *design entry*. We will describe two design entry methods: using schematic capture and writing source code in a hardware description language.

Schematic Capture

A logic circuit can be defined by drawing logic gates and interconnecting them with wires. A CAD tool for entering a designed circuit in this way is called a *schematic capture* tool. The word *schematic* refers to a diagram of a circuit in which circuit elements, such as logic gates, are depicted as graphical symbols and connections between circuit elements are drawn as lines.

A schematic capture tool uses the graphics capabilities of a computer and a computer mouse to allow the user to draw a schematic diagram. To facilitate inclusion of gates in the schematic, the tool provides a collection of graphical symbols that represent gates of various types with different numbers of inputs. This collection of symbols is called a *library*. The gates in the library can be imported into the user's schematic, and the tool provides a graphical way of interconnecting the gates to create a logic network.

Any subcircuits that have been previously created can be represented as graphical symbols and included in the schematic. In practice it is common for a CAD system user to create a circuit that includes within it other smaller circuits. This methodology is known as *hierarchical design* and provides a good way of dealing with the complexities of large circuits.

The schematic-capture method is simple to use, but becomes awkward when large circuits are involved. A better method for dealing with large circuits is to write source code using a hardware description language to represent the circuit.

Hardware Description Languages

A *hardware description language (HDL)* is similar to a typical computer programming language except that an HDL is used to describe hardware rather than a program to be executed on a computer. Many commercial HDLs are available. Some are proprietary, meaning that they are provided by a particular company and can be used to implement circuits only in the technology offered by that company. We will not discuss the proprietary HDLs in this book. Instead, we will focus on a language that is supported by virtually all vendors that provide digital hardware technology and is officially endorsed as an *Institute of Electrical and Electronics Engineers (IEEE)* standard. The IEEE is a worldwide organization that promotes technical activities to the benefit of society in general. One of its activities involves the development of standards that define how certain technological concepts can be used in a way that is suitable for a large body of users.

Two HDLs are IEEE standards: *Verilog HDL* and *VHDL (Very High Speed Integrated Circuit Hardware Description Language)*. Both languages are in widespread use in the industry. We use VHDL in this book, but a Verilog version of the book is also available from the same publisher. Although the two languages differ in many ways, the choice of using one or the other when studying logic circuits is not particularly important, because both offer similar

features. Concepts illustrated in this book using VHDL can be directly applied when using Verilog.

In comparison to performing schematic capture, using VHDL offers a number of advantages. Because it is supported by most organizations that offer digital hardware technology, VHDL provides design *portability*. A circuit specified in VHDL can be implemented in different types of chips and with CAD tools provided by different companies, without having to change the VHDL specification. Design portability is an important advantage because digital circuit technology changes rapidly. By using a standard language, the designer can focus on the functionality of the desired circuit without being overly concerned about the details of the technology that will eventually be used for implementation.

Design entry of a logic circuit is done by writing VHDL code. Signals in the circuit can be represented as variables in the source code, and logic functions are expressed by assigning values to these variables. VHDL source code is plain text, which makes it easy for the designer to include within the code documentation that explains how the circuit works. This feature, coupled with the fact that VHDL is widely used, encourages sharing and reuse of VHDL-described circuits. This allows faster development of new products in cases where existing VHDL code can be adapted for use in the design of new circuits.

Similar to the way in which large circuits are handled in schematic capture, VHDL code can be written in a modular way that facilitates hierarchical design. Both small and large logic circuit designs can be efficiently represented in VHDL code. We will introduce VHDL in Section 2.10.

2.9.2 LOGIC SYNTHESIS

Synthesis is the process of generating a logic circuit from an initial specification that may be given in the form of a schematic diagram or code written in a hardware description language. Synthesis CAD tools generate efficient implementations of circuits from such specifications.

The process of translating, or *compiling*, VHDL code into a network of logic gates is part of synthesis. The output is a set of logic expressions that describe the logic functions needed to realize the circuit.

Regardless of what type of design entry is used, the initial logic expressions produced by the synthesis tools are not likely to be in an optimal form because they reflect the designer's input to the CAD tools. It is impossible for

a designer to manually produce optimal results for large circuits. So, one of the important tasks of the synthesis tools is to manipulate the user's design to automatically generate an equivalent, but better circuit.

The measure of what makes one circuit better than another depends on the particular needs of a design project and the technology chosen for implementation. Earlier in this chapter we suggested that a good circuit might be one that has the lowest cost. There are other possible optimization goals, which are motivated by the type of hardware technology used for implementation of the circuit. We discuss implementation technologies in Appendix B.

The performance of a synthesized circuit can be assessed by physically constructing the circuit and testing it. But, its behavior can also be evaluated by means of simulation.

2.9.3 FUNCTIONAL SIMULATION

A circuit represented in the form of logic expressions can be simulated to verify that it will function as expected. The tool that performs this task is called a *functional simulator*. It uses the logic expressions (often referred to as equations) generated during synthesis, and assumes that these expressions will be implemented with perfect gates through which signals propagate instantaneously. The simulator requires the user to specify valuations of the circuit's inputs that should be applied during simulation. For each valuation, the simulator evaluates the outputs produced by the expressions. The results of simulation are usually provided in the form of a timing diagram which the user can examine to verify that the circuit operates as required.

2.9.4 PHYSICAL DESIGN

After logic synthesis the next step in the design flow is to determine exactly how to implement the circuit on a given chip. This step is often called *physical design*. As we discuss in Appendix B, there are several different technologies that may be used to implement logic circuits. The physical design tools map a circuit specified in the form of logic expressions into a realization that makes use of the resources available on the target chip. They determine the placement of specific logic elements, which are not necessarily simple gates of the type we have encountered so far. They also determine the wiring connections that have to be made between these elements to implement the desired circuit.

2.9.5 TIMING ANALYSIS

Logic gates and other logic elements are implemented with electronic circuits, and these circuits cannot perform their function with zero delay. When the values of inputs to the circuit change, it takes a certain amount of time before a corresponding change occurs at the output. This is called a *propagation delay* of the circuit. The propagation delay consists of two kinds of delays. Each logic element needs some time to generate a valid output signal whenever there are changes in the values of its inputs. In addition to this delay, there is a delay caused by signals that must propagate along wires that connect various logic elements. The combined effect is that real circuits exhibit delays, which has a significant impact on their speed of operation.

A *timing analyzer* evaluates the expected delays of a designed logic circuit. Its results can be used to determine if the generated circuit meets the timing requirements of the specification for the design. If the requirements are not met, the designer can ask the physical design tools to try again by indicating specific timing constraints that have to be met. If this does not succeed, then the designer has to try different optimizations in the synthesis step or else improve the initial design that is presented to the synthesis tools.

2.9.6 CIRCUIT IMPLEMENTATION

Having ascertained that the designed circuit meets all requirements of the specification, the circuit is implemented on an actual chip. If a custom-manufactured chip is created for this design, then this step is called *chip fabrication*. But if a programmable hardware device is used, then this step is called chip *configuration* or *programming*. Various types of chip technologies are described in Appendix B.

2.9.7 COMPLETE DESIGN FLOW

The CAD tools discussed above are the essential parts of a CAD system. The complete design flow that we discussed is illustrated in Figure 2.35. This has been just a brief introductory discussion. A full presentation of the CAD tools is given in Chapter 10.

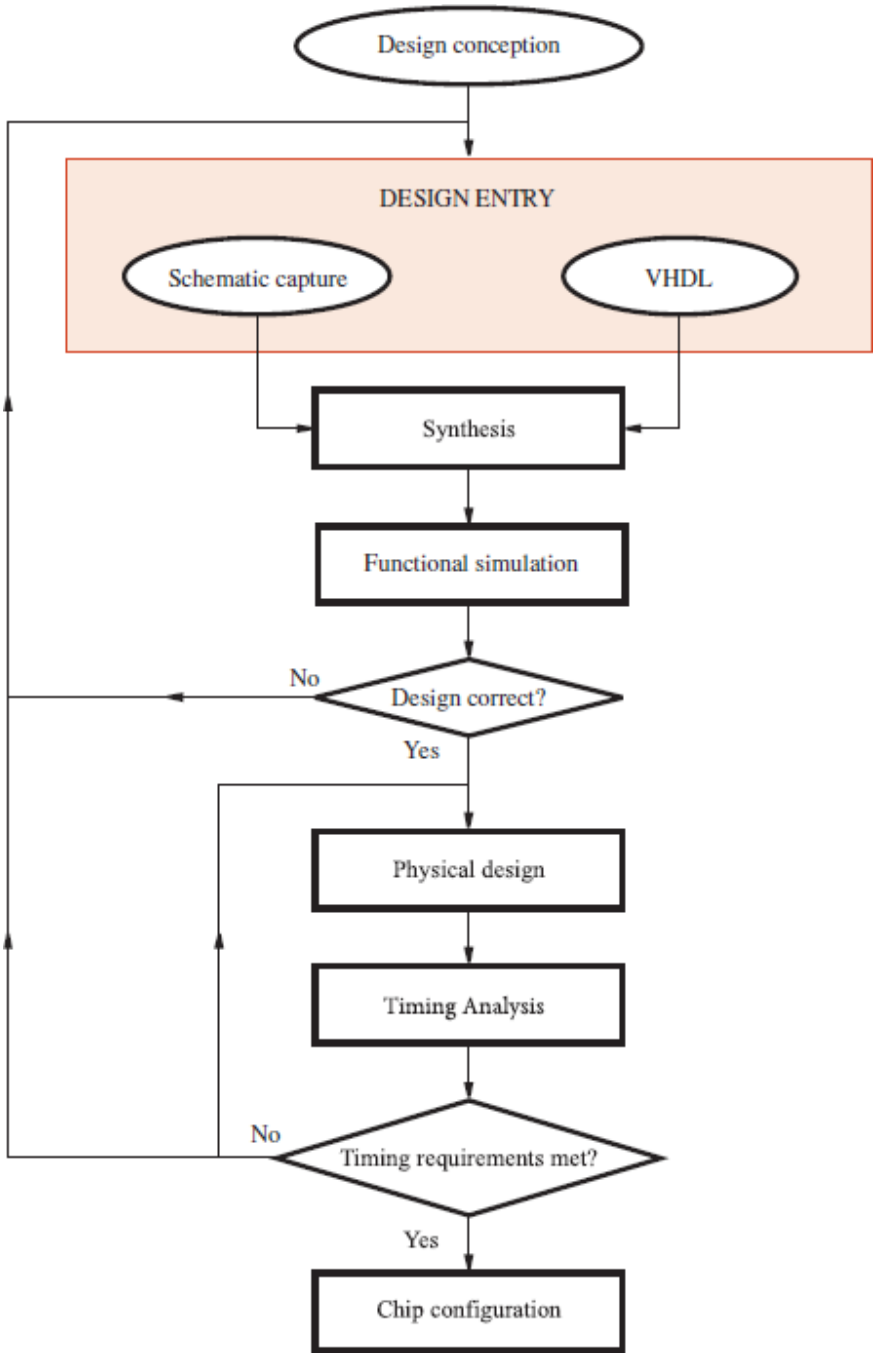


Figure 2.35 A typical CAD system.

At this point the reader should have some appreciation for what is involved when using CAD tools. However, the tools can be fully appreciated only when they are used firsthand. We strongly encourage the reader to obtain access to suitable CAD tools and implement some examples of circuits by using these tools. Two examples of commonly used CAD tools are the Quartus tools available from Intel Corporation and the ISE tools provided by

Xilinx Corporation. Both of these CAD systems can be obtained free of charge for educational use from their respective corporations' websites.

2.10 Introduction to VHDL

In the 1980s rapid advances in integrated circuit technology led to efforts to develop standard design practices for digital circuits. VHDL was developed as a part of that effort and became an industry standard language for describing digital circuits. In 1987, VHDL was adopted as an IEEE standard, called IEEE 1076. A revised standard was adopted in 1993 and called IEEE 1164. The standard was subsequently updated in 2000 and 2002.

VHDL was originally intended to serve two main purposes. First, it was used as a documentation language for describing the structure of complex digital circuits. As an official IEEE standard, VHDL provided a common way of documenting circuits designed by numerous designers. Second, VHDL provided features for modeling the behavior of a digital circuit, which allowed its use as input to software programs that were then used to simulate the circuit's operation.

In recent years, in addition to its use for documentation and simulation, VHDL has also become popular for use in design entry in CAD systems. The CAD tools are used to synthesize the VHDL code into a hardware implementation of the described circuit. In this book our main use of VHDL will be for synthesis.

VHDL is a complex, sophisticated language. Learning all of its features is a daunting task. However, for use in synthesis only a subset of these features is important. To simplify the presentation, we will focus the discussion on the features of VHDL that are actually used in the examples in the book. The material presented should be sufficient to allow the reader to design a wide range of circuits. The reader who wishes to learn the complete VHDL language can refer to one of the specialized texts.

VHDL is introduced in several stages throughout the book. Our general approach will be to introduce particular features only when they are relevant to the design topics covered in that part of the text. In Appendix A we provide a concise summary of the VHDL features covered in the book. The reader will find it convenient to refer to that material from time to time. In the

remainder of this chapter, we discuss the most basic concepts needed to write simple VHDL code.

Representation of Digital Circuits in VHDL

When using CAD tools to synthesize a logic circuit, the designer can provide the initial description of the circuit in several different ways, as we explained in Section 2.9.1. One efficient way is to write this description in the form of VHDL source code. The VHDL compiler translates this code into a logic circuit.

VHDL allows the designer to describe a desired circuit in a number of ways. One possibility is to use VHDL statements that describe the structure of the circuit using operators corresponding to logic gates, such as AND and OR. A larger circuit is defined by writing code that connects these logic gates together. This approach is referred to as the *structural* representation of logic circuits. Another possibility is to describe a circuit more abstractly, by using logic expressions and VHDL constructs that define the desired behavior of the circuit, but not its actual structure in terms of gates. This is called the *behavioral* representation.

2.10.1 Structural Specification of Logic Circuits

Each logic signal in the circuit is represented in VHDL code as a data object. Just as the variables declared in any high-level programming language have associated types, such as integers or characters, data objects in VHDL can be of various types. The original VHDL standard, IEEE 1076, includes a data type called *BIT*. An object of this type is well suited for representing digital signals because BIT objects can have only two values, 0 and 1. We will give two examples below that use the BIT type and then introduce another data type that is commonly used for logic circuits.

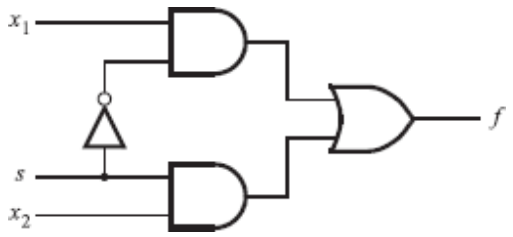


Figure 2.36 The logic circuit for a multiplexer.

We will use an example to illustrate how to write simple VHDL source code. Consider the multiplexer circuit from Figure 2.33*b*, which is reproduced in Figure 2.36. If we wish to write VHDL code to represent this circuit, the first step is to declare the input and output signals. This is done using a construct called an *entity*. An appropriate entity for this example appears in Figure 2.37. An entity must be assigned a name; we have chosen the name *example1* for this first example. The input and output signals for the entity are called its *ports*, and they are identified by the keyword PORT. This name derives from the electrical jargon in which the word *port* refers to an input or output connection to an electronic circuit. Each port has an associated *mode* that specifies whether it is an input (IN) to the entity or an output (OUT) from the entity. Each port represents a signal; hence, it has an associated type. The entity *example1* has four ports in total. The first three, x_1, x_2, and *s*, are input signals of type BIT. The port named *f* is an output of type BIT.

```
ENTITY example1 IS
    PORT ( x1, x2, s  : IN   BIT ;
           f          : OUT  BIT ) ;
END example1 ;
```

Figure 2.37 VHDL entity declaration for the circuit in Figure 2.36.

In Figure 2.37 we have used simple signal names *x1*, *x2*, *s*, and *f* for the entity's ports. Similar to most computer programming languages, VHDL has rules that specify which characters are allowed in signal names. A simple guideline is that signal names can include any letter or number, as well as the underscore character `_'. There are two caveats: a signal name must begin with a letter, and a signal name cannot be a VHDL keyword.

An entity specifies the input and output signals for a circuit, but it does not give any details as to what the circuit represents. The circuit's functionality must be specified with a VHDL construct called an *architecture*. An architecture for our example appears in Figure 2.38. It must be given a name, and we have chosen the name *LogicFunc*. Although the name can be any text string, it is sensible to assign a name that is meaningful to the designer. In this case we have chosen the name *LogicFunc* because the architecture specifies the functionality of the design using a logic expression. VHDL has built-in support for the following Boolean operators: AND, OR, NOT, NAND, NOR, XOR, and XNOR (complement of XOR). Following the BEGIN keyword, our architecture specifies, using the VHDL signal assignment operator <=, that output *f* should be assigned the result of the logic expression on the right-

hand side of the operator. Because VHDL does not assume any precedence of logic operators, parentheses are used in the expression. One might expect that an assignment statement such as

```
ARCHITECTURE LogicFunc OF example1 IS
BEGIN
    f <= (NOT s AND x1) OR (s AND x2) ;
END LogicFunc ;
```

Figure 2.38 VHDL architecture for the entity in Figure 2.37.

$$f <= \text{NOT } s \text{ AND } x1 \text{ OR } s \text{ AND } x2$$

would have implied parentheses

$$f <= (\text{NOT } s \text{ AND } x1) \text{ OR } (s \text{ AND } x2)$$

But for VHDL code this assumption is not true. In fact, without the parentheses the VHDL compiler would produce a compile-time error for this expression.

Complete VHDL code for our example is given in Figure 2.39. This example has illustrated that a VHDL source code file has two main sections: an entity and an architecture. A simple analogy for what each section represents is that the entity is equivalent to a symbol in a schematic diagram and the architecture specifies the logic circuitry inside the symbol.

```
ENTITY example1 IS
    PORT ( x1, x2, s  : IN   BIT ;
           f          : OUT  BIT ) ;
END example1 ;

ARCHITECTURE LogicFunc OF example1 IS
BEGIN
    f <= (NOT s AND x1) OR (s AND x2) ;
END LogicFunc ;
```

Figure 2.39 Complete VHDL code for the circuit in Figure 2.36.

A second example of VHDL code is given in Figure 2.40. This circuit has four input signals, called $x1$, $x2$, $x3$, and $x4$, and two output signals, named f and g. A logic expression is assigned to each output. A logic circuit produced by the VHDL compiler for this example is shown in Figure 2.41.

```
ENTITY example2 IS
    PORT ( x1, x2, x3, x4   : IN     BIT ;
           f, g             : OUT  BIT ) ;
END example2 ;

ARCHITECTURE LogicFunc OF example2 IS
BEGIN
    f <= (x1 AND x3) OR (x2 AND x4) ;
    g <= (x1 OR NOT x3) AND (NOT x2 OR x4) ;
END LogicFunc ;
```

Figure 2.40 VHDL code for a four-input function.

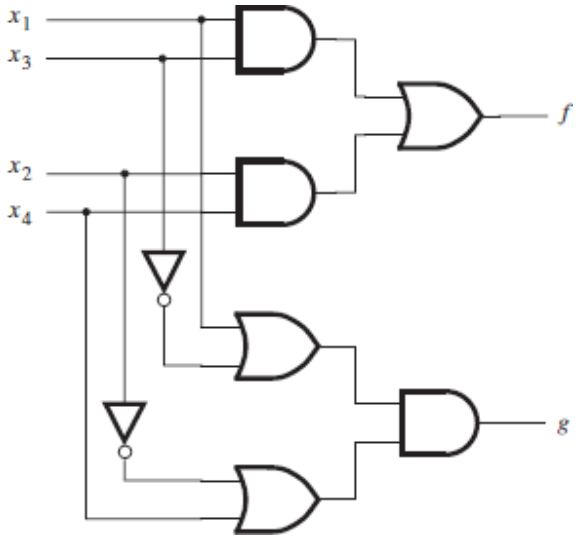


Figure 2.41 Logic circuit for the code in Figure 2.40.

The preceding two examples indicate that one way to assign a value to a signal in VHDL code is by means of a logic expression. In VHDL terminology, this type of logic expression is called a *signal assignment statement*. VHDL also supports several other types of assignment statements and many other features that are useful for describing circuits that are much more complex.

Using VHDL Data Types

In addition to the type BIT, a commonly used VHDL data type for logic circuit design is called *STD_LOGIC*. As its name implies, STD_LOGIC is meant to serve as the standard data type for representation of logic signals. Figure 2.42 gives the VHDL code from Figure 2.39 modified to use the STD_LOGIC type instead of the BIT type.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY example3 IS
    PORT ( x1, x2, s  : IN     STD_LOGIC ;
           f          : OUT    STD_LOGIC ) ;
END example3 ;

ARCHITECTURE LogicFunc OF example3 IS
BEGIN
    f <= (NOT s AND x1) OR (s AND x2) ;
END LogicFunc ;
```

Figure 2.42 Modified VHDL code from Figure 2.39 using the STD_LOGIC data type.

To use the STD_LOGIC type, VHDL code must include the two lines given at the beginning of the figure. These statements serve as directives to the VHDL compiler. They are needed because the original VHDL standard, IEEE 1076, did not include the STD_LOGIC type. The way that the new type was added to the language, in the IEEE 1164 standard, was to provide the definition of STD_LOGIC as a set of files that can be included with VHDL code when compiled. The set of files is called a *library*. The purpose of the first line in Figure 2.42 is to declare that the code will make use of the IEEE library.

As we said earlier, signals with the type BIT can only assume the value 0 or 1. Signals that have the STD_LOGIC type can take a number of different values, with the most useful values for describing logic functions being 0, 1, Z, and -. We show how the logic value Z, which represents *high-impedance*, is useful in Appendix B. The - value represents an unspecified logic value, known as a *don't-care condition*. We discuss don't-care conditions later in this chapter.

In Figure 2.42 the second line of code tells the VHDL compiler that we wish to use the STD_LOGIC type. The definition of this type is encapsulated in a special type of file known as a *package*. The package is named std_logic_1164. It is possible to instruct the VHDL compiler to use only a subset of the package, but the normal use is to specify the word *all* to indicate that the entire package is of interest, as we have done in the figure.

For the examples of VHDL code given in this book, we will almost always declare signals with the type STD_LOGIC.

2.10.2 BEHAVIORAL SPECIFICATION OF LOGIC CIRCUITS

Using logic expressions is a simple approach for writing VHDL code. But higher levels of abstraction can often be used to advantage. Consider again the multiplexer circuit of Figure 2.36. The circuit can be described in words by saying that $f = x_1$ if $s = 0$ and $f = x_2$ if $s = 1$. In VHDL, this behavior can be defined with the *if-else* statement

```
IF s = '0' THEN
   f <= x1 ;
ELSE
   f <= x2 ;
END IF ;
```

The complete code is given in Figure 2.43. The first line illustrates how a comment can be inserted. The if-else statement is an example of a VHDL *sequential assignment* statement. We will introduce other sequential assignment statements, such as loop statements, in Chapters 3 and 4.

```
-- Behavioral specification
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY example4 IS
    PORT ( x1, x2, s  : IN   STD_LOGIC ;
           f          : OUT  STD_LOGIC ) ;
END example4 ;

ARCHITECTURE Behavior OF example4 IS
BEGIN
    PROCESS ( x1, x2, s )
    BEGIN
        IF s = '0' THEN
            f <= x1 ;
        ELSE
            f <= x2 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 2.43 Behavioral specification of the circuit in Figure 2.36.

VHDL syntax requires that sequential assignment statements, such as the if-else statement, be placed inside a construct called a *process*. The process begins with the PROCESS keyword, followed by a parenthesized list of signals, called the *sensitivity list*. The sensitivity list tells the VHDL compiler which signals can affect the output signals produced by the process. In Figure

2.43 the sensitivity list comprises the multiplexer data inputs, x_1 and x_2, and the select input s.

Behavioral specification of a logic circuit defines only its behavior. CAD synthesis tools use this specification to construct the actual circuit. The detailed structure of the synthesized circuit will depend on the technology used.

2.10.3 HIERARCHICAL VHDL CODE

The examples of VHDL code given so far include just a single entity. For larger designs, it is often convenient to create a hierarchical structure in the VHDL code, in which there is a *top-level* entity that includes multiple instances of *lower-level* entities. To see how hierarchical VHDL code can be written, consider the circuit in Figure 2.44. This circuit comprises two lower-level entities: the adder entity that we described in Figure 2.12, and the entity that drives a 7-segment display which we showed in Figure 2.34. The purpose of the circuit is to generate the arithmetic sum of the two inputs x and y, using the *adder* entity, and then to show the resulting decimal value on the 7-segment display.

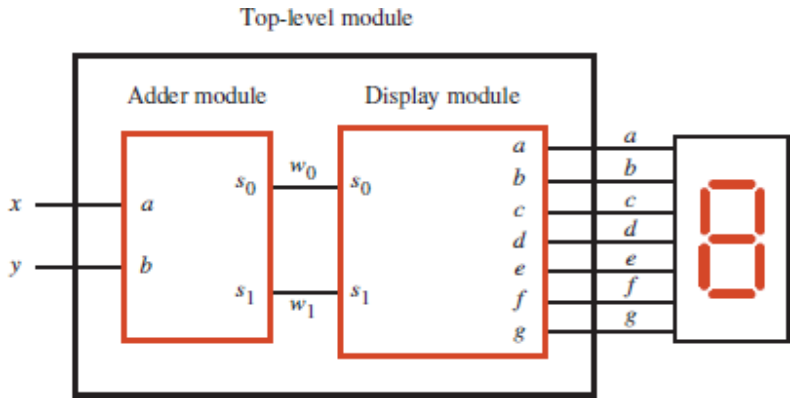


Figure 2.44 A logic circuit with two entities.

VHDL code for the *adder* entity from Figure 2.12 and the *display* entity from Figure 2.34 is given in Figures 2.45 and 2.46, respectively. For the *adder* entity signal assignment statements are used to specify the two-bit sum s_1 s_0. The code for the *display* entity includes signal assignment statements that correspond to the logic expressions for each of the seven outputs of the display circuit, which are given in Section 2.8.3. The statement

```
-- A circuit for driving a 7-segment display
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY display IS
    PORT ( s1, s0              : IN    STD_LOGIC ;
           a, b, c, d, e, f, g : OUT   STD_LOGIC ) ;
END display ;

ARCHITECTURE Behavior OF display IS
BEGIN
    a <= NOT s0 ;
    b <= '1' ;
    c <= NOT s1 ;
    d <= NOT s0 ;
    e <= NOT s0 ;
    f <= NOT s1 AND NOT s0 ;
    g <= s1 AND NOT s0 ;
END Behavior ;
```

Figure 2.46 VHDL specification of the circuit in Figure 2.34.

```
b <= '1' ;
```

assigns the output *b* of the display entity to have the constant value 1. We discuss the specification of numbers in VHDL code in Chapter 3.

The top-level VHDL entity, named *adder_display*, is given in Figure 2.47. This entity has the inputs *x* and *y*, and the outputs *a*, …, *g*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder_display IS
    PORT ( x, y              : IN    STD_LOGIC ;
           a, b, c, d, e, f, g : OUT  STD_LOGIC ) ;
END adder_display ;

ARCHITECTURE Structure OF adder_display IS
    SIGNAL w1, w0 : STD_LOGIC ;
    COMPONENT adder IS
        PORT ( a, b   : IN    STD_LOGIC ;
               s1, s0 : OUT   STD_LOGIC ) ;
    END COMPONENT ;
    COMPONENT display IS
        PORT ( s1, s0            : IN    STD_LOGIC ;
               a, b, c, d, e, f, g : OUT  STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    U1: adder PORT MAP ( x, y, w1, w0 ) ;
    U2: display PORT MAP ( w1, w0, a, b, c, d, e, f, g ) ;
END Structure ;
```

Figure 2.47 Hierarchical VHDL code for the circuit in Figure 2.44.

The architecture body has the name *Structure*. We chose this name because the style of code in which a circuit is described in a hierarchical fashion, by connecting together subcircuits, is usually called the *structural* style. In previous examples of VHDL code, all signals were declared as ports in the entity declaration. As shown in Figure 2.47, signals can also be declared preceding the BEGIN keyword in the architecture body. The declaration

```
SIGNAL w1, w0 : STD_LOGIC ;
```

is needed because the signals w_1 and w_0 are neither inputs nor outputs of the circuit in Figure 2.44. Since these signals cannot be declared as input or output ports in the VHDL code, they have to be declared as (internal) *signals*.

The next statement in the code is a *component declaration* statement for the *adder* entity. This statement uses syntax similar to that in an entity declaration. It allows the *adder* entity to be used as a component (subcircuit) in the architecture body. Similarly, the *display* entity is also declared as a component. The statement

```
U1: adder PORT MAP ( x, y, w1, w0 ) ;
```

instantiates the *adder* entity from Figure 2.45 as a subcircuit. The subcircuit is given an *instance name*, *U*1, which can be any valid VHDL name. In this instantiation statement the signals attached to the ports of the *adder* component are listed in the same order as those in Figure 2.45. Thus, the input ports *x* and *y* of the top-level entity in Figure 2.47 are connected to the first two ports of *adder*, which are named *a* and *b*. The order in which signals are listed in the instantiation statement determines which signal is connected to each port in the component. The instantiation statement also attaches the last two ports of the *adder* component, which are its outputs, to the signals *w*1 and *w*0 in the top-level entity.

```
-- An adder circuit
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder IS
    PORT ( a, b     : IN    STD_LOGIC ;
           s1, s0   : OUT   STD_LOGIC ) ;
END adder ;

ARCHITECTURE Behavior OF adder IS
BEGIN
    s1 <= a AND b ;
    s0 <= a XOR b ;
END Behavior ;
```

Figure 2.45 VHDL specification of the circuit in Figure 2.12.

The statement

```
U2: display PORT MAP ( w1, w0, a, b, c, d, e, f, g ) ;
```

instantiates the other subcircuit in our circuit. Here, the wires *w*1 and *w*0, which have already been connected to the outputs of the *adder* component, are attached to the corresponding input ports of the *display* component. The *display* component's output ports are attached to the *a*, …, *g* output ports of the top-level entity.

2.10.4 HOW *NOT* TO WRITE VHDL CODE

When learning how to use VHDL or other hardware description languages, the tendency for the novice is to write code that resembles a computer program, containing many variables and loops. It is difficult to determine

what logic circuit the CAD tools will produce when synthesizing such code. This book contains more than 100 examples of complete VHDL code that represent a wide range of logic circuits. In these examples the code is easily related to the described logic circuit. The reader is advised to adopt the same style of code. A good general guideline is to assume that if the designer cannot readily determine what logic circuit is described by the VHDL code, then the CAD tools are not likely to synthesize the circuit that the designer is trying to model.

Once complete VHDL code is written for a particular design, the reader is encouraged to analyze the resulting circuit synthesized by the CAD tools. Much can be learned about VHDL, logic circuits, and logic synthesis through this process.

2.11 Minimization and Karnaugh Maps

In a number of our examples we have used algebraic manipulation to find a reduced-cost implementation of a function in either sum-of-products or product-of-sums form. In these examples, we made use of the rules, theorems, and properties of Boolean algebra that were introduced in Section 2.5. For example, we often used the distributive property, DeMorgan's theorem, and the combining property. In general, it is not obvious when to apply these theorems and properties to find a minimum-cost circuit, and it is often tedious and impractical to do so. This section introduces a more manageable approach, called, the *Karnaugh map*, which provides a systematic way of producing a minimum-cost logic expression.

The key to the Karnaugh map approach is that it allows the application of the combining property 14*a*, or 14*b*, as judiciously as possible. To understand how it works, consider the function f in Figure 2.48. The canonical sum-of-products expression for f consists of minterms m_0, m_2, m_4, m_5, and m_6, so that

Row number	x_1	x_2	x_3	f
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 2.48 The function $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

$$f = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3$$

The combining property 14*a* allows us to replace two minterms that differ in the value of only one variable with a single product term that does not include that variable at all. For example, both m_0 and m_2 include $\bar{x}_1$ and $\bar{x}_3$, but they differ in the value of x_2 because m_0 includes $\bar{x}_2$ while m_2 includes x_2. Thus

$$\begin{aligned}\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 &= \bar{x}_1(\bar{x}_2 + x_2)\bar{x}_3 \\ &= \bar{x}_1 \cdot 1 \cdot \bar{x}_3 \\ &= \bar{x}_1\bar{x}_3\end{aligned}$$

Hence, m_0 and m_2 can be replaced by the single product term $\bar{x}_1\bar{x}_3$. Similarly, m_4 and m_6 differ only in the value of x_2 and can be combined using

$$\begin{aligned}x_1\bar{x}_2\bar{x}_3 + x_1x_2\bar{x}_3 &= x_1(\bar{x}_2 + x_2)\bar{x}_3 \\ &= x_1 \cdot 1 \cdot \bar{x}_3 \\ &= x_1\bar{x}_3\end{aligned}$$

Now the two newly generated terms, $\bar{x}_1\bar{x}_3$ and $x_1\bar{x}_3$, can be combined further as

$$\begin{aligned}\bar{x}_1\bar{x}_3 + x_1\bar{x}_3 &= (\bar{x}_1 + x_1)\bar{x}_3 \\ &= 1 \cdot \bar{x}_3 \\ &= \bar{x}_3\end{aligned}$$

These optimization steps indicate that we can replace the four minterms m_0, m_2, m_4, and m_6 with the single product term $\bar{x}_3$. In other words, the minterms

m_0, m_2, m_4, and m_6 are all *included* in the term $\bar{x}_3$. The remaining minterm in f is m_5. It can be combined with m_4, which gives

$$x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 = x_1\bar{x}_2$$

Recall that theorem 7*b* in Section 2.5 indicates that

$$m_4 = m_4 + m_4$$

which means that we can use the minterm m_4 twice—to combine with minterms m_0, m_2, and m_6 to yield the term $\bar{x}_3$ as explained above and also to combine with m_5 to yield the term $x_1\bar{x}_2$.

We have now accounted for all the minterms in f; hence all five input valuations for which $f = 1$ are covered by the minimum-cost expression

$$f = \bar{x}_3 + x_1\bar{x}_2$$

The expression has the product term $\bar{x}_3$ because $f = 1$ when $x_3 = 0$ regardless of the values of x_1 and x_2. The four minterms m_0, m_2, m_4, and m_6 represent all possible minterms for which $x_3 = 0$; they include all four valuations, 00, 01, 10, and 11, of variables x_1 and x_2. Thus if $x_3 = 0$, then it is guaranteed that $f = 1$. This may not be easy to see directly from the truth table in Figure 2.48, but it is obvious if we write the corresponding valuations grouped together:

	x_1	x_2	x_3
m_0	0	0	0
m_2	0	1	0
m_4	1	0	0
m_6	1	1	0

In a similar way, if we look at m_4 and m_5 as a group of two

	x_1	x_2	x_3
m_4	1	0	0
m_5	1	0	1

it is clear that when $x_1 = 1$ and $x_2 = 0$, then $f = 1$ regardless of the value of x_3.

The preceding discussion suggests that it would be advantageous to devise a method that allows easy discovery of groups of minterms for which $f = 1$ that can be combined into single terms. The Karnaugh map is a useful vehicle for this purpose.

The *Karnaugh map* is an alternative to the truth-table form for representing a function. The map consists of *cells* that correspond to the rows of the truth table. Consider the two-variable example in Figure 2.49. Part (*a*) depicts the truth-table form, where each of the four rows is identified by a minterm. Part (*b*) shows the Karnaugh map, which has four cells. The columns of the map are labeled by the value of x_1, and the rows are labeled by x_2. This labeling leads to the locations of minterms as shown in the figure. Compared to the truth table, the advantage of the Karnaugh map is that it allows easy recognition of minterms that can be combined using property 14*a* from Section 2.5. Minterms in any two cells that are adjacent, either in the same row or the same column, can be combined. For example, the minterms m_2 and m_3 can be combined as

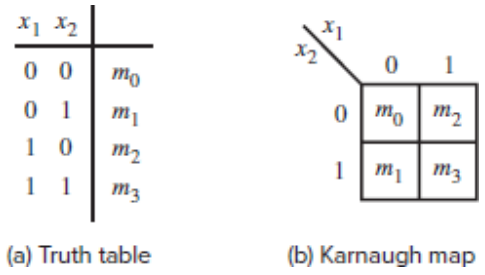


Figure 2.49 Location of two-variable minterms.

$$\begin{aligned} m_2 + m_3 &= x_1\bar{x}_2 + x_1x_2 \\ &= x_1(\bar{x}_2 + x_2) \\ &= x_1 \cdot 1 \\ &= x_1 \end{aligned}$$

The Karnaugh map is not just useful for combining pairs of minterms. As we will see in several larger examples, the Karnaugh map can be used directly to derive a minimum-cost circuit for a logic function.

Two-Variable Map

A Karnaugh map for a two-variable function is given in Figure 2.50. It corresponds to the function *f* of Figure 2.19. The value of *f* for each valuation

of the variables x_1 and x_2 is indicated in the corresponding cell of the map. Because a 1 appears in both cells of the bottom row and these cells are adjacent, there exists a single product term that can cause f to be equal to 1 when the input variables have the values that correspond to either of these cells. To indicate this fact, we have circled the cell entries in the map. Rather than using the combining property formally, we can derive the product term intuitively. Both of the cells are identified by $x_2 = 1$, but $x_1 = 0$ for the left cell and $x_1 = 1$ for the right cell. Thus if $x_2 = 1$, then $f = 1$ regardless of whether x_1 is equal to 0 or 1. The product term representing the two cells is simply x_2.

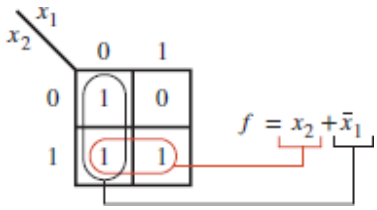


Figure 2.50 The function of Figure 2.19.

Similarly, $f = 1$ for both cells in the first column. These cells are identified by $x_1 = 0$. Therefore, they lead to the product term $\bar{x}_1$. Since this takes care of all instances where $f = 1$, it follows that the minimum-cost realization of the function is

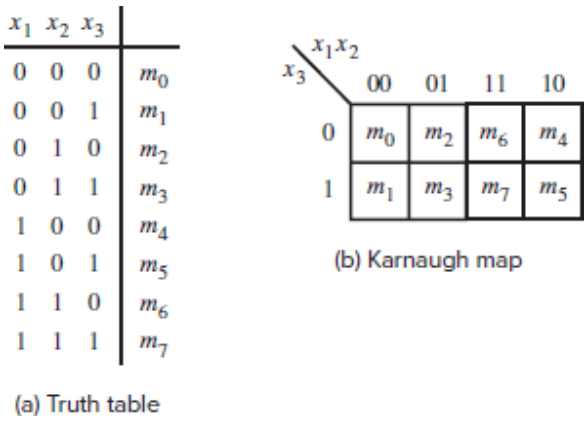
$$f = x_2 + \bar{x}_1$$

Evidently, to find a minimum-cost implementation of a given function, it is necessary to find the smallest number of product terms that produce a value of 1 for all cases where $f = 1$. Moreover, the cost of these product terms should be as low as possible. Note that a product term that covers two adjacent cells is cheaper to implement than a term that covers only a single cell. For our example once the two cells in the bottom row have been covered by the product term x_2, only one cell (top left) remains. Although it could be covered by the term $\bar{x}_1\bar{x}_2$, it is better to combine the two cells in the left column to produce the product term $\bar{x}_1$ because this term is cheaper to implement.

Three-Variable Map

A three-variable Karnaugh map is constructed by placing 2 two-variable maps side by side. Figure 2.51*a* lists all of the three-variable minterms, and

part (*b*) of the figure indicates the locations of these minterms in the Karnaugh map. In this case each valuation of x_1 and x_2 identifies a column in the map, while the value of x_3 distinguishes the two rows. To ensure that minterms in the adjacent cells in the map can always be combined into a single product term, the adjacent cells must differ in the value of only one variable. Thus the columns are identified by the sequence of (x_1, x_2) values of 00, 01, 11, and 10, rather than the more obvious 00, 01, 10, and 11. This makes the second and third columns different only in variable x_1. Also, the first and the fourth columns differ only in variable x_1, which means that these columns can be considered as being adjacent. The reader may find it useful to visualize the map as a rectangle folded into a cylinder where the left and the right edges in Figure 2.51*b* are made to touch. (A sequence of codes, or valuations, where consecutive codes differ in one variable only is known as the *Gray code*. This code is used for a variety of purposes, some of which will be encountered later in the book.)



x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

(a) Truth table

x_3 \ x_1x_2	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

(b) Karnaugh map

Figure 2.51 Location of three-variable minterms.

Figure 2.52*a* represents the function of Figure 2.23 in Karnaugh-map form. To synthesize this function, it is necessary to cover the four 1s in the map as efficiently as possible. It is not difficult to see that two product terms suffice. The first covers the 1s in the top row, which are represented by the term $x_1\bar{x}_3$. The second term is $\bar{x}_2x_3$, which covers the 1s in the bottom row. Hence the function is implemented as

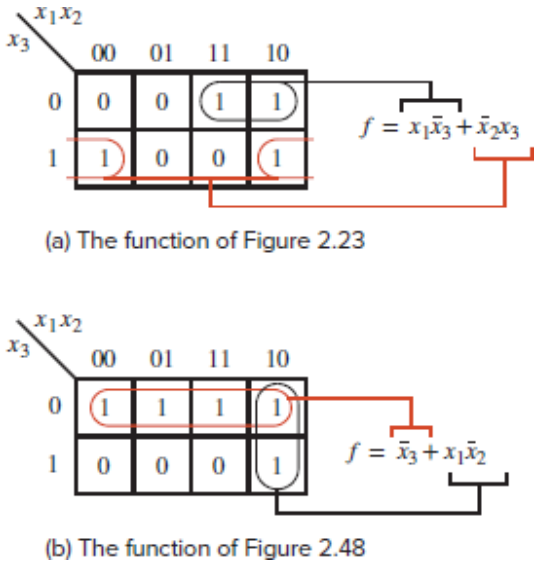


Figure 2.52 Examples of three-variable Karnaugh maps.

$$f = x_1\bar{x}_3 + \bar{x}_2x_3$$

which describes the circuit obtained in Figure 2.24*a*.

In a three-variable map, it is possible to combine cells to produce product terms that correspond to a single cell, two adjacent cells, or a group of four adjacent cells. Realization of a group of four adjacent cells using a single product term is illustrated in Figure 2.52*b*, using the function from Figure 2.48. The four cells in the top row correspond to the (x_1, x_2, x_3) valuations 000, 010, 110, and 100. As we discussed before, this indicates that if $x_3 = 0$, then $f = 1$ for all four possible valuations of x_1 and x_2, which means that the only requirement is that $x_3 = 0$. Therefore, the product term $\bar{x}_3$ represents these four cells. The remaining 1, corresponding to minterm m_5, is best covered by the term $x_1\bar{x}_2$, obtained by combining the two cells in the right most column. The complete realization of f is

$$f = \bar{x}_3 + x_1\bar{x}_2$$

It is also possible to have a group of eight 1s in a three-variable map. This is the trivial case of a function where $f = 1$ for all valuations of input variables; in other words, f is equal to the constant 1.

The Karnaugh map provides a simple mechanism for generating the product terms that should be used to implement a given function. A product term must include only those variables that have the same value for all cells in

the group represented by this term. If the variable is equal to 1 in the group, it appears uncomplemented in the product term; if it is equal to 0, it appears complemented. Each variable that is sometimes 1 and sometimes 0 in the group does not appear in the product term.

Four-Variable Map

A four-variable map is constructed by placing 2 three-variable maps together to create four rows in the same fashion as we used 2 two-variable maps to form the four columns in a three-variable map. Figure 2.53 shows the structure of the four-variable map and the location of minterms. We have included in this figure another frequently used way of designating the rows and columns. As shown in red, it is sufficient to indicate the rows and columns for which a given variable is equal to 1. Thus $x_1 = 1$ for the two right most columns, $x_2 = 1$ for the two middle columns, $x_3 = 1$ for the bottom two rows, and $x_4 = 1$ for the two middle rows.

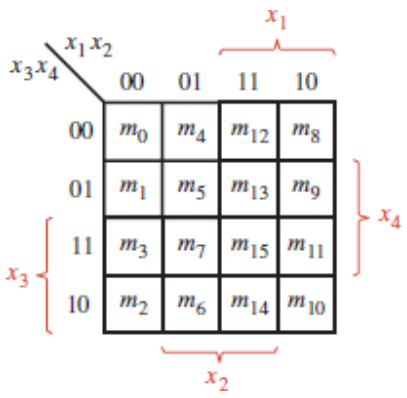


Figure 2.53 A four-variable Karnaugh map.

Figure 2.54 gives four examples of four-variable functions. The function f_1 has a group of four 1s in adjacent cells in the bottom two rows, for which $x_2 = 0$ and $x_3 = 1$—they are represented by the product term $\bar{x}_2x_3$. This leaves the two 1s in the second row to be covered, which can be accomplished with the term $x_1\bar{x}_3x_4$. Hence the minimum-cost implementation of the function is

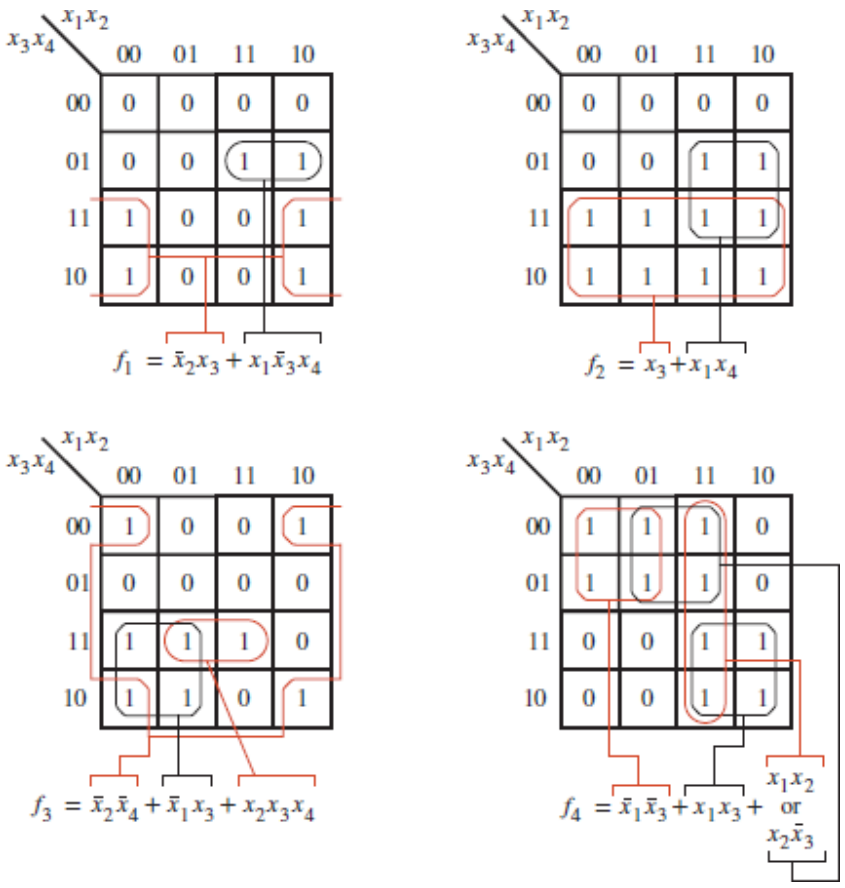


Figure 2.54 Examples of four-variable Karnaugh maps.

$$f_1 = \bar{x}_2x_3 + x_1\bar{x}_3x_4$$

The function f_2 includes a group of eight 1s that can be implemented by a single term, x_3. Again, the reader should note that if the remaining two 1s were implemented as a group of two, the result would be the product term $x_1\bar{x}_3x_4$. Implementing these 1s as a part of a group of four 1s, as shown in the figure, gives the less expensive product term $x_1\ x_4$.

Just as the left and the right edges of the map are adjacent in terms of the assignment of the variables, so are the top and the bottom edges. Indeed, the four corners of the map are adjacent to each other and thus can form a group of four 1s, which may be implemented by the product term $\bar{x}_2\bar{x}_4$. This case is depicted by the function f_3. In addition to this group of 1s, there are four other 1s that must be covered to implement f_3. This can be done as shown in the figure.

In all examples that we have considered so far, a unique solution exists that leads to a minimum-cost circuit. The function f_4 provides an example where there is some choice. The groups of four 1s in the top-left and bottom-right corners of the map are realized by the terms $\bar{x}_1\bar{x}_3$ and $x_1\ x_3$, respectively. This leaves the two 1s that correspond to the term $x_1x_2\bar{x}_3$. But these two 1s can be realized more economically by treating them as a part of a group of four 1s. They can be included in two different groups of four, as shown in the figure. One choice leads to the product term $x_1\ x_2$, and the other leads to $x_2\bar{x}_3$. Both of these terms have the same cost; hence it does not matter which one is chosen in the final circuit. Note that the complement of x_3 in the term $x_2\bar{x}_3$ does not imply an increased cost in comparison with $x_1\ x_2$, because this complement must be generated anyway to produce the term $\bar{x}_1\bar{x}_3$, which is included in the implementation.

We can use 2 four-variable maps to construct a five-variable map. It is easy to imagine a structure where one map is directly behind the other, and they are distinguished by $x_5 = 0$ for one map and $x_5 = 1$ for the other map. Since such a structure is awkward to draw, we can simply place the two maps side by side as shown in Figure 2.55. For the logic function given in this example, two groups of four 1s appear in the same place in both four-variable maps; hence their realization does not depend on the value of x_5. The same is true for the two groups of two 1s in the second row. The 1 in the top-right corner appears only in the right map, where $x_5 = 1$; it is a part of the group of two 1s realized by the term $x_1\bar{x}_2\bar{x}_3x_5$. Note that in this map we left blank those cells for which $f = 0$, to make the figure more readable. We will do likewise in a number of maps that follow.

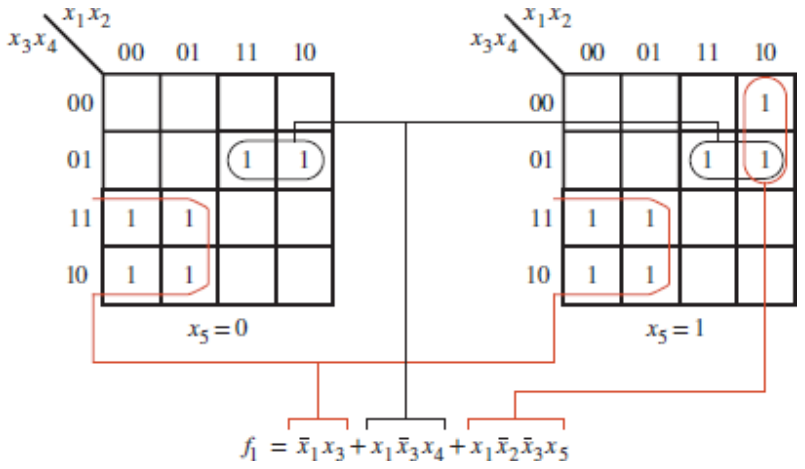


Figure 2.55 A five-variable Karnaugh map.

Five-Variable Map

Using a five-variable map is obviously more awkward than using maps with fewer variables. Extending the Karnaugh map concept to more variables is not useful from the practical point of view. This is not troublesome, because practical synthesis of logic functions is done with CAD tools that perform the necessary minimization automatically. Although Karnaugh maps are occasionally useful for designing small logic circuits, our main reason for introducing the Karnaugh maps is to provide a simple vehicle for illustrating the ideas involved in the minimization process.

2.12 Strategy for Minimization

For the examples in the preceding section, we used an intuitive approach to decide how the 1s in a Karnaugh map should be grouped together to obtain the minimum-cost implementation of a given function. Our intuitive strategy was to find as few as possible and as large as possible groups of 1s that cover all cases where the function has a value of 1. Each group of 1s has to comprise cells that can be represented by a single product term. The larger the group of 1s, the fewer the number of variables in the corresponding product term. This approach worked well because the Karnaugh maps in our examples were small. For larger logic functions, which have many variables, such intuitive approach is unsuitable. Instead, we must have an organized method for deriving a minimum-cost implementation. In this section we will introduce a possible method, which is similar to the techniques that are automated in CAD tools. To illustrate the main ideas, we will use Karnaugh maps.

2.12.1 Terminology

A huge amount of research work has gone into the development of techniques for synthesis of logic functions. The results of this research have been published in numerous papers. To facilitate the presentation of the results, certain terminology has evolved that avoids the need for using highly

descriptive phrases. We define some of this terminology in the following paragraphs because it is useful for describing the minimization process.

Literal

A given product term consists of some number of variables, each of which may appear either in uncomplemented or complemented form. Each appearance of a variable, either uncomplemented or complemented, is called a *literal*. For example, the product term $x_1\bar{x}_2x_3$ has three literals, and the term $\bar{x}_1x_3\bar{x}_4x_6$ has four literals.

Implicant

A product term that indicates the input valuation(s) for which a given function is equal to 1 is called an *implicant* of the function. The most basic implicants are the minterms, which we introduced in Section 2.6.1. For an *n*-variable function, a minterm is an implicant that consists of *n* literals.

Consider the three-variable function in Figure 2.56. There are 11 implicants for this function. This includes the five minterms: $\bar{x}_1\bar{x}_2\bar{x}_3, \bar{x}_1\bar{x}_2x_3, \bar{x}_1x_2\bar{x}_3, \bar{x}_1x_2x_3$, and $x_1\ x_2\ x_3$. Then there are the implicants that correspond to all possible pairs of minterms that can be combined, namely, $\bar{x}_1\bar{x}_2$ (m_0 and m_1), $\bar{x}_1\bar{x}_3$ (m_0 and m_2), $\bar{x}_1x_3$ (m_1 and m_3), $\bar{x}_1x_2$ (m_2 and m_3), and $x_2\ x_3$ (m_3 and m_7). Finally, there is one implicant that covers a group of four minterms, which consists of a single literal $\bar{x}_1$.

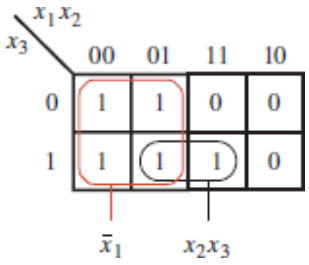


Figure 2.56 Three-variable function $f(x_1, x_2, x_3) = \sum m\ (0, 1, 2, 3, 7)$.

Prime Implicant

An implicant is called a *prime implicant* if it cannot be combined into another implicant that has fewer literals. Another way of stating this definition is to say that it is impossible to delete any literal in a prime implicant and still have a valid implicant.

In Figure 2.56 there are two prime implicants: $\bar{x}_1$ and $x_2\,x_3$. It is not possible to delete a literal in either of them. Doing so for $\bar{x}_1$ would make it disappear. For $x_2\,x_3$, deleting a literal would leave either x_2 or x_3. But x_2 is not an implicant because it includes the valuation $(x_1, x_2, x_3) = 110$ for which $f = 0$, and x_3 is not an implicant because it includes $(x_1, x_2, x_3) = 101$ for which $f = 0$. Another way of thinking about prime implicants is that they represent "the largest groups of 1s" that can be circled in the Karnaugh map.

Cover

A collection of implicants that account for all valuations for which a given function is equal to 1 is called a *cover* of that function. A number of different covers exist for most functions. Obviously, a set of all minterms for which $f = 1$ is a cover. It is also apparent that a set of all prime implicants is a cover.

A cover defines a particular implementation of the function. In Figure 2.56 a cover consisting of minterms leads to the expression

$$f = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_1x_2x_3 + x_1x_2x_3$$

Another valid cover is given by the expression

$$f = \bar{x}_1\bar{x}_2 + \bar{x}_1x_2 + x_2x_3$$

The cover comprising the prime implicants is

$$f = \bar{x}_1 + x_2x_3$$

While all of these expressions represent the function f correctly, the cover consisting of prime implicants leads to the lowest-cost implementation.

Cost

In Section 2.6.1 we suggested that a good indication of the cost of a logic circuit is the number of gates plus the total number of inputs to all gates in the circuit. We will use this definition of cost throughout the book. But we will assume that primary inputs, namely, the input variables, are available in both true and complemented forms at zero cost. Thus the expression

$$f = x_1\bar{x}_2 + x_3\bar{x}_4$$

has a cost of nine because it can be implemented using two AND gates and one OR gate, with six inputs to the AND and OR gates.

If an inversion is needed inside a circuit, then the corresponding NOT gate and its input are included in the cost. For example, the expression

$$g = \left(\overline{x_1\bar{x}_2 + x_3}\right)(\bar{x}_4 + x_5)$$

is implemented using two AND gates, two OR gates, and one NOT gate to complement $(x_1\bar{x}_2 + x_3)$, with nine inputs. Hence the total cost is 14.

2.12.2 MINIMIZATION PROCEDURE

We have seen that it is possible to implement a given logic function with various circuits. These circuits may have different structures and different costs. When designing a logic circuit, there are usually certain criteria that must be met. One such criterion is likely to be the cost of the circuit, which we considered in the previous discussion. In general, the larger the circuit, the more important the cost issue becomes. In this section we will assume that the main objective is to obtain a minimum-cost circuit.

In the previous subsection we concluded that the lowest-cost implementation is\linebreak achieved when the cover of a given function consists of prime implicants. The question then is how to determine the minimum-cost subset of prime implicants that will cover the function. Some prime implicants may have to be included in the cover, while for others there may be a choice. If a prime implicant includes a minterm for which $f = 1$ that is not included in any other prime implicant, then it must be included in the cover and is called an *essential prime implicant*. In the example in Figure 2.56, both prime implicants are essential. The term $x_2\ x_3$ is the only prime implicant that covers the minterm m_7, and $\bar{x}_1$ is the only one that covers the minterms m_0, m_1, and m_2. Notice that the minterm m_3 is covered by both of these prime implicants. The minimum-cost realization of the function is

$$f = \bar{x}_1 + x_2x_3$$

We will now present several examples in which there is a choice as to which prime implicants to include in the final cover. Consider the four-variable function in Figure 2.57. There are five prime implicants: $\bar{x}_1x_3, \bar{x}_2x_3, x_3\bar{x}_4, \bar{x}_1x_2x_4$, and $x_2\bar{x}_3x_4$. The essential ones (highlighted in red) are $\bar{x}_2x_3$

(because of m_{14}), and $x_2\bar{x}_3x_4$ (because of m_{13}). They must be included in the cover. These three prime implicants cover all minterms for which $f = 1$ except m_7. It is clear that m_7 can be covered by either $\bar{x}_1x_3$ or $\bar{x}_1x_2x_4$. Because $\bar{x}_1x_3$ has a lower cost, it is chosen for the cover. Therefore, the minimum-cost realization is

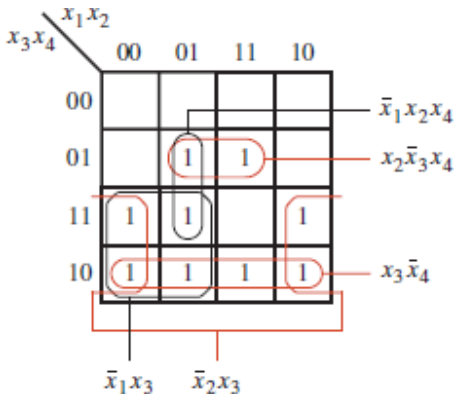


Figure 2.57 Four-variable function $f(x_1, \ldots, x_4) = \sum m\,(2, 3, 5, 6, 7, 10, 11, 13, 14)$.

$$f = \bar{x}_2x_3 + x_3\bar{x}_4 + x_2\bar{x}_3x_4 + \bar{x}_1x_3$$

From the preceding discussion, the process of finding a minimum-cost circuit involves the following steps:

1. Generate all prime implicants for the given function *f*.
2. Find the set of essential prime implicants.
3. If the set of essential prime implicants covers all valuations for which $f = 1$, then this set is the desired cover of *f*. Otherwise, determine the nonessential prime implicants that should be added to form a complete minimum-cost cover.

The choice of nonessential prime implicants to be included in the cover is governed by the cost considerations. This choice is often not obvious. Indeed, for large functions there may exist many possibilities, and some *heuristic* approach (i.e., an approach that considers only a subset of possibilities but gives good results most of the time) has to be used. One such approach is to arbitrarily select one nonessential prime implicant and include it in the cover and then determine the rest of the cover. Next, another cover is determined assuming that this prime implicant is not in the cover. The costs of the resulting covers are compared, and the less-expensive cover is chosen for implementation.

We can illustrate the process by using the function in Figure 2.58. Of the six prime implicants, only $\bar{x}_3\bar{x}_4$ is essential. Consider next $x_1x_2\bar{x}_3$ and assume first that it will be included in the cover. Then the remaining three minterms, m_{10}, m_{11}, and m_{15}, will require two more prime implicants to be included in the cover. A possible implementation is

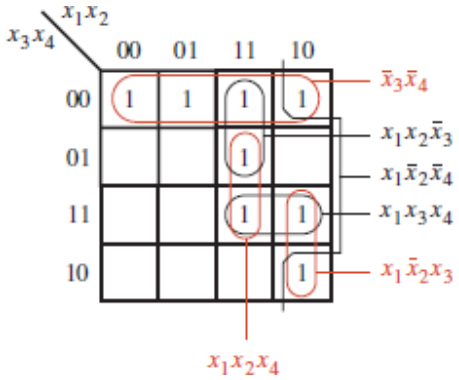


Figure 2.58 The function $f(x_1, \ldots, x_4) = \sum m\,(0, 4, 8, 10, 11, 12, 13, 15)$.

$$f = \bar{x}_3\bar{x}_4 + x_1x_2\bar{x}_3 + x_1x_3x_4 + x_1\bar{x}_2x_3$$

The second possibility is that $x_1x_2\bar{x}_3$ is not included in the cover. Then $x_1\ x_2\ x_4$ becomes essential because there is no other way of covering m_{13}. Because $x_1\ x_2\ x_4$ also covers m_{15}, only m_{10} and m_{11} remain to be covered, which can be achieved with $x_1\bar{x}_2x_3$. Therefore, the alternative implementation is

$$f = \bar{x}_3\bar{x}_4 + x_1x_2x_4 + x_1\bar{x}_2x_3$$

Clearly, this implementation is a better choice.

Sometimes there may not be any essential prime implicants at all. An example is given in Figure 2.59. Choosing any of the prime implicants and first including it, then excluding it from the cover leads to two alternatives of equal cost. One includes the prime implicants indicated in black, which yields

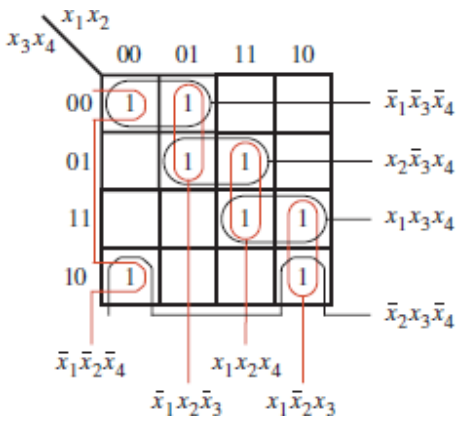


Figure 2.59 The function $f(x_1, \ldots, x_4) = \sum m\,(0, 2, 4, 5, 10, 11, 13, 15)$.

$$f = \bar{x}_1\bar{x}_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_1x_3x_4 + \bar{x}_2x_3\bar{x}_4$$

The other includes the prime implicants indicated in red, which yields

$$f = \bar{x}_1\bar{x}_2\bar{x}_4 + \bar{x}_1x_2\bar{x}_3 + x_1x_2x_4 + x_1\bar{x}_2x_3$$

This procedure can be used to find minimum-cost implementations of both small and large logic functions. For our small examples it was convenient to use Karnaugh maps to determine the prime implicants of a function and then choose the final cover. Other techniques based on the same principles are much more suitable for use in CAD tools; we will introduce such techniques in Chapter 8.

The previous examples have been based on the sum-of-products form. We will next illustrate that the same concepts apply for the product-of-sums form.

2.13 Minimization of Product-of-Sums Forms

Now that we know how to find the minimum-cost sum-of-products (SOP) implementations of functions, we can use the same techniques and the principle of duality to obtain minimum-cost product-of-sums (POS) implementations. In this case it is the maxterms for which $f = 0$ that have to be combined into sum terms that are as large as possible. Again, a sum term is considered larger if it covers more maxterms, and the larger the term, the less costly it is to implement.

Figure 2.60 depicts the same function as Figure 2.56 depicts. There are three maxterms that must be covered: M_4, M_5, and M_6. They can be covered by two sum terms shown in the figure, leading to the following implementation:

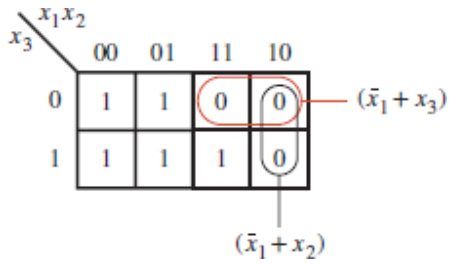


Figure 2.60 POS minimization of $f(x_1, x_2, x_3) = \Pi M (4, 5, 6)$.

$$f = (\bar{x}_1 + x_2)(\bar{x}_1 + x_3)$$

A circuit corresponding to this expression has two OR gates and one AND gate, with two inputs for each gate. Its cost is greater than the cost of the equivalent SOP implementation derived in Figure 2.56, which requires only one OR gate and one AND gate.

The function from Figure 2.57 is reproduced in Figure 2.61. The maxterms for which $f = 0$ can be covered as shown, leading to the expression

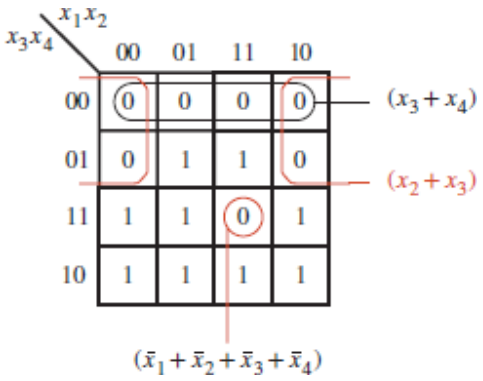


Figure 2.61 POS minimization of $f(x_1, \ldots, x_4) = \Pi M (0, 1, 4, 8, 9, 12, 15)$.

$$f = (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)$$

This expression represents a circuit with three OR gates and one AND gate. Two of the OR gates have two inputs, and the third has four inputs; the AND gate has three inputs. Assuming that both the complemented and

uncomplemented versions of the input variables x_1 to x_4 are available at no extra cost, the cost of this circuit is 15. This compares favorably with the SOP implementation derived from Figure 2.57, which requires five gates and 13 inputs at a total cost of 18.

In general, as we already know from Section 2.6.1, the SOP and POS implementations of a given function may or may not entail the same cost. The reader is encouraged to find the POS implementations for the functions in Figures 2.58 and 2.59 and compare the costs with the SOP forms.

We have shown how to obtain minimum-cost POS implementations by finding the largest sum terms that cover all maxterms for which $f = 0$. Another way of obtaining the same result is by finding a minimum-cost SOP implementation of the complement of f. Then we can apply DeMorgan's theorem to this expression to obtain the simplest POS realization because $f = \bar{\bar{f}}$. For example, the simplest SOP implementation of $\bar{f}$ in Figure 2.60 is

$$\bar{f} = x_1\bar{x}_2 + x_1\bar{x}_3$$

Complementing this expression using DeMorgan's theorem yields

$$\begin{aligned} f = \bar{\bar{f}} &= \overline{x_1\bar{x}_2 + x_1\bar{x}_3} \\ &= \overline{x_1\bar{x}_2} \cdot \overline{x_1\bar{x}_3} \\ &= (\bar{x}_1 + x_2)(\bar{x}_1 + x_3) \end{aligned}$$

which is the same result as obtained above.

Using this approach for the function in Figure 2.61 gives

$$\bar{f} = \bar{x}_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2x_3x_4$$

Complementing this expression produces

$$\begin{aligned} f = \bar{\bar{f}} &= \overline{\bar{x}_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2x_3x_4} \\ &= \overline{\bar{x}_2\bar{x}_3} \cdot \overline{\bar{x}_3\bar{x}_4} \cdot \overline{x_1x_2x_3x_4} \\ &= (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4) \end{aligned}$$

which matches the previously derived implementation.

2.14 Incompletely Specified Functions

In digital systems it often happens that certain input conditions can never occur. For example, suppose that x_1 and x_2 control two interlocked switches such that both switches cannot be closed at the same time. Thus the only three possible states of the switches are that both switches are open or that one switch is open and the other switch is closed. Namely, the input valuations $(x_1, x_2) = 00$, 01, and 10 are possible, but 11 is guaranteed not to occur. Then we say that $(x_1, x_2) = 11$ is a *don't-care condition*, meaning that a circuit with x_1 and x_2 as inputs can be designed by ignoring this condition. A function that has don't-care condition(s) is said to be *incompletely specified*.

Don't-care conditions, or *don't-cares* for short, can be used to advantage in the design of logic circuits. Since these input valuations will never occur, the designer may assume that the function value for these valuations is either 1 or 0, whichever is more useful in trying to find a minimum-cost implementation. Figure 2.62 illustrates this idea. The required function has a value of 1 for minterms m_2, m_4, m_5, m_6, and m_{10}. Assuming the above-mentioned interlocked switches, the x_1 and x_2 inputs will never be equal to 1 at the same time; hence the minterms m_{12}, m_{13}, m_{14}, and m_{15} can all be used as don't-cares. The don't-cares are denoted by the letter d in the map. Using the shorthand notation, the function f is specified as

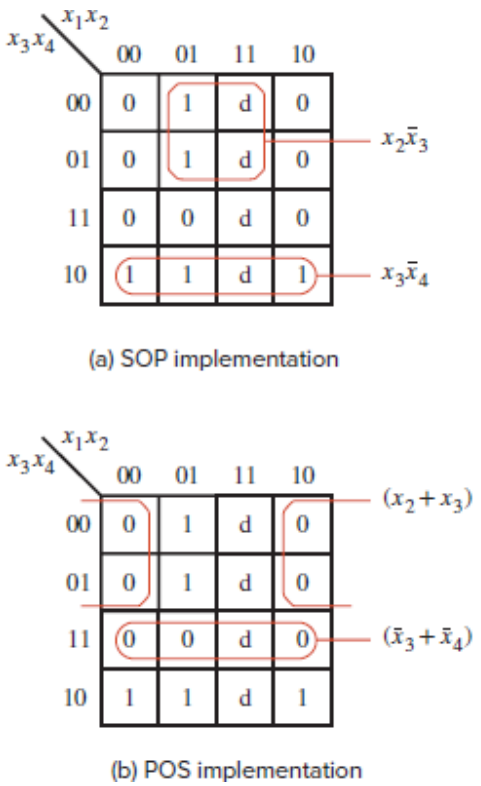


Figure 2.62 Two implementations of the function $f(x_1, \ldots, x_4) = \sum m\,(2, 4, 5, 6, 10) + D\,(12, 13, 14, 15)$.

$$f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$

where D is the set of don't-cares.

Part (*a*) of the figure indicates the best sum-of-products implementation. To form the largest possible groups of 1s, thus generating the lowest-cost prime implicants, it is necessary to assume that the don't-cares D_{12}, D_{13}, and D_{14} (corresponding to minterms m_{12}, m_{13}, and m_{14}) have the value of 1 while D_{15} has the value of 0. Then there are only two prime implicants, which provide a complete cover of f. The resulting implementation is

$$f = x_2\bar{x}_3 + x_3\bar{x}_4$$

Part (*b*) shows how the best product-of-sums implementation can be obtained. The same values are assumed for the don't cares. The result is

$$f = (x_2 + x_3)(\bar{x}_3 + \bar{x}_4)$$

The freedom in choosing the value of don't-cares leads to greatly simplified realizations. If we were to naively exclude the don't-cares from the synthesis of the function, by assuming that they always have a value of 0, the resulting SOP expression would be

$$f = \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 x_3 \bar{x}_4 + \bar{x}_2 x_3 \bar{x}_4$$

and the POS expression would be

$$f = (x_2 + x_3)(\bar{x}_3 + \bar{x}_4)(\bar{x}_1 + \bar{x}_2)$$

Both of these expressions have higher costs than the expressions obtained with a more appropriate assignment of values to don't-cares.

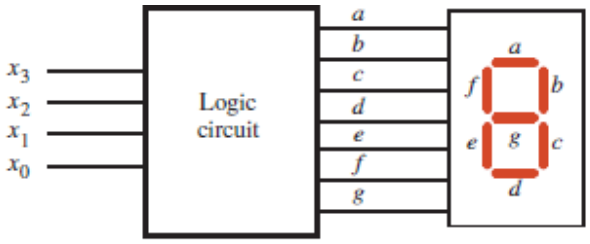
Although don't-care values can be assigned arbitrarily, an arbitrary assignment may not lead to a minimum-cost implementation of a given function. If there are k don't-cares, then there are 2^k possible ways of assigning 0 or 1 values to them. In the Karnaugh map we can usually see how best to do this assignment to find the simplest implementation.

In the example above, we chose the don't-cares D_{12}, D_{13}, and D_{14} to be equal to 1 and D_{15} equal to 0 for both the SOP and POS implementations. Thus the derived expressions represent the same function, which could also be specified as $\sum m(2, 4, 5, 6, 10, 12, 13, 14)$. Assigning the same values to the don't-cares for both SOP and POS implementations is not always a good choice. Sometimes it may be advantageous to give a particular don't-care the value 1 for SOP implementation and the value 0 for POS implementation, or vice versa. In such cases the optimal SOP and POS expressions will represent different functions, but these functions will differ only for the valuations that correspond to these don't-cares. Example 2.26 in Section 2.17 illustrates this possibility.

Using interlocked switches to illustrate how don't-care conditions can occur in a real system may seem to be somewhat contrived. A more practical example is shown below, and in future chapters we will encounter many examples of don't-cares that occur in the course of practical design of digital circuits.

Example 2.15

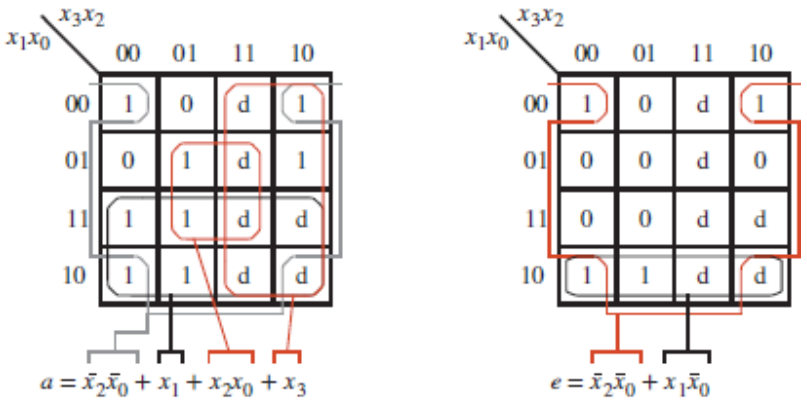
In Section 2.8.3 we designed a logic circuit that displays the decimal value of a two-bit number on a seven-segment display. In this example we will design a similar circuit, except that the input will be a four-bit number $X = x_3\, x_2\, x_1\, x_0$ that represents the decimal values 0, 1, …, 9. Using four bits to represent decimal digits in this way is often referred to as the *binary coded decimal* (BCD) representation. We discuss BCD numbers in detail in Chapter 3. The circuit is depicted in Figure 2.63*a*. Part (*b*) of the figure gives a truth table for each of the seven outputs *a*, *b*, …, *g*, that control the display. It also indicates how the display should appear for each value of *X*. Since *X* is restricted to decimal digits, then the values of *X* from 1010 to 1111 are not used. These entries are omitted from the truth table in Figure 2.63*b* and can be treated as don't-cares in the design of the circuit.



(a) Logic circuit and 7-segment display

	x_3	x_2	x_1	x_0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1

(b) Truth table



(c) The Karnaugh maps for outputs a and e.

Figure 2.63 Using don't-care minterms when displaying BCD numbers.

To derive logic expressions for the outputs a to g, it is useful to draw Karnaugh maps. Figure 2.63*c* gives Karnaugh maps for the functions a and e. For the function a, the best result is obtained by setting all six don't-care cells in the map to the value of 1, which gives $a = \bar{x}_2\bar{x}_0 + x_1 + x_2x_0 + x_3$. However, for the function e a better choice is to set only two of the don't-care cells,

corresponding to $x_3\ x_2\ x_1\ x_0 = 1010$ and $x_3\ x_2\ x_1\ x_0 = 1110$, to 1. The other don't-care cells should be set to 0 for this function to yield the minimum-cost expression $e = \bar{x}_2\bar{x}_0 + x_1\bar{x}_0$.

2.15 Multiple-Output Circuits

As illustrated in Example 2.15, in practical digital systems it is often necessary to implement a number of functions that are part of a larger logic circuit. Instead of implementing each of these functions separately, it may be possible to share some of the gates needed in the implementation of individual functions. For example, in Figure 2.63 the AND gate that produces $\bar{x}_2\bar{x}_0$ could be shared for use in both functions *a* and *e*.

Example 2.16

An example of gate sharing is given in Figure 2.64. Two functions, f_1 and f_2, of the same variables are to be implemented. The minimum-cost implementations for these functions are obtained as shown in parts (*a*) and (*b*) of the figure. This results in the expressions

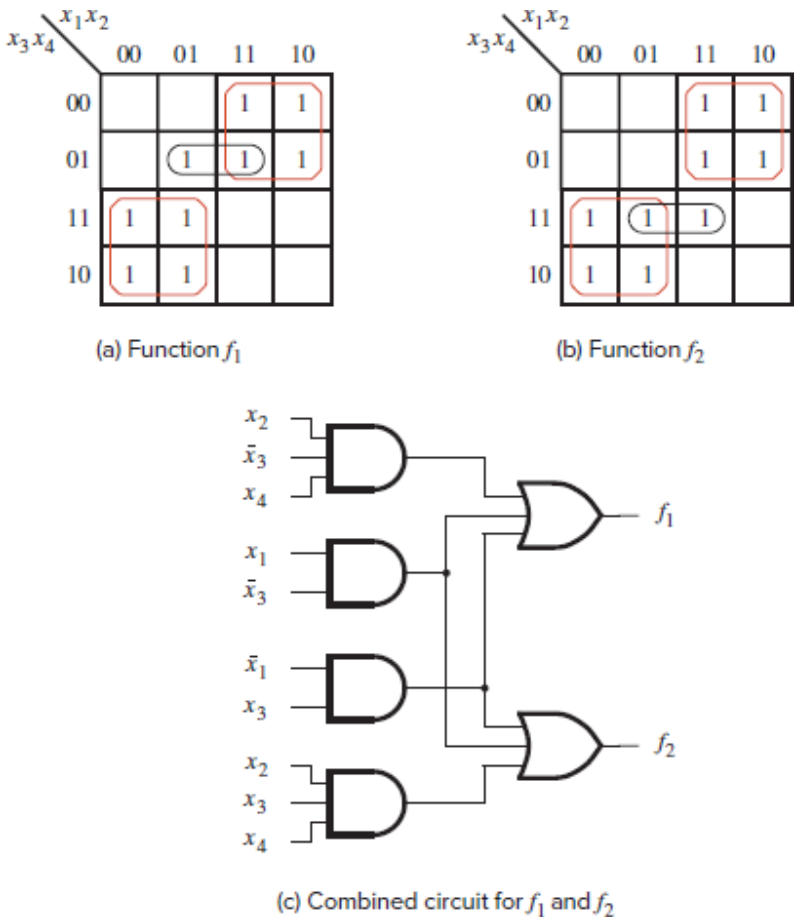


Figure 2.64 An example of multiple-output synthesis.

$$f_1 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2\bar{x}_3x_4$$
$$f_2 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2x_3x_4$$

The cost of f_1 is four gates and 10 inputs, for a total of 14. The cost of f_2 is the same. Thus the total cost is 28 if both functions are implemented by separate circuits. A less-expensive realization is possible if the two circuits are combined into a single circuit with two outputs. Because the first two product terms are identical in both expressions, the AND gates that implement them need not be duplicated. The combined circuit is shown in Figure 2.64*c*. Its cost is six gates and 16 inputs, for a total of 22.

In this example we reduced the overall cost by finding minimum-cost realizations of f_1 and f_2 and then sharing the gates that implement the common product terms. This strategy does not necessarily always work the best, as the next example shows.

Example 2.17

Figure 2.65 shows two functions to be implemented by a single circuit. Minimum-cost realizations of the individual functions f_3 and f_4 are obtained from parts (*a*) and (*b*) of the figure.

$$f_3 = \bar{x}_1 x_4 + x_2 x_4 + \bar{x}_1 x_2 x_3$$
$$f_4 = x_1 x_4 + \bar{x}_2 x_4 + \bar{x}_1 x_2 x_3 \bar{x}_4$$

None of the AND gates can be shared, which means that the cost of the combined circuit would be six AND gates, two OR gates, and 21 inputs, for a total of 29.

But several alternative realizations are possible. Instead of deriving the expressions for f_3 and f_4 using only prime implicants, we can look for other implicants that may be shared advantageously in the combined realization of the functions. Figure 2.65*c* shows the best choice of implicants, which yields the realization

$$f_3 = x_1 x_2 x_4 + \bar{x}_1 x_2 x_3 \bar{x}_4 + \bar{x}_1 x_4$$
$$f_4 = x_1 x_2 x_4 + \bar{x}_1 x_2 x_3 \bar{x}_4 + \bar{x}_2 x_4$$

The first two implicants are identical in both expressions. The resulting circuit is given in Figure 2.65*d*. It has the cost of six gates and 17 inputs, for a total of 23.

Example 2.18

In Example 2.16 we sought the best SOP implementation for the functions f_1 and f_2 in Figure 2.64. We will now consider the POS implementation of the same functions. The minimum-cost POS expressions for f_1 and f_2 are

$$f_1 = (\bar{x}_1 + \bar{x}_3)(x_1 + x_2 + x_3)(x_1 + x_3 + x_4)$$
$$f_2 = (x_1 + x_3)(\bar{x}_1 + x_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_3 + x_4)$$

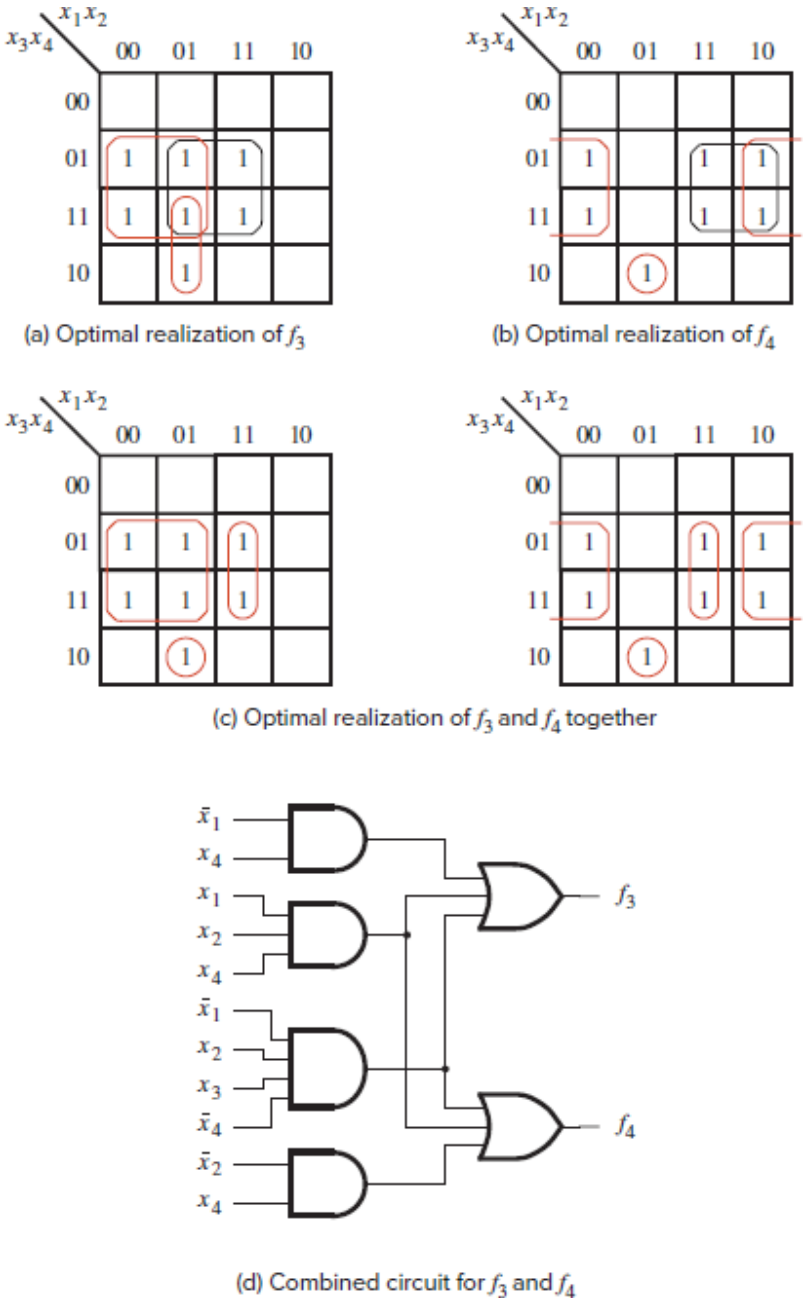

x_1x_2
x_3x_4
00 01 11 10
(a) Optimal realization of f_3
(b) Optimal realization of f_4
(c) Optimal realization of f_3 and f_4 together
$\bar{x}_1$ x_4 x_1 x_2 x_4 $\bar{x}_1$ x_2 x_3 $\bar{x}_4$ $\bar{x}_2$ x_4
f_3
f_4
(d) Combined circuit for f_3 and f_4

Figure 2.65 Another example of multiple-output synthesis.

There are no common sum terms in these expressions that could be shared in the implementation. Moreover, from the Karnaugh maps in Figure 2.64, it is apparent that there is no sum term (covering the cells where $f_1 = f_2 = 0$) that can be profitably used in realizing both f_1 and f_2. Thus the best choice is to implement each function separately, according to the preceding expressions. Each function requires three OR gates, one AND gate, and 11 inputs.

Therefore, the total cost of the circuit that implements both functions is 30. This realization is costlier than the SOP realization derived in Example 2.16.

Example 2.19

consider now the POS realization of the functions f_3 and f_4 in Figure 2.65. The minimum-cost POS expressions for f_3 and f_4 are

$$f_3 = (x_3 + x_4)(x_2 + x_4)(\bar{x}_1 + x_4)(\bar{x}_1 + x_2)$$
$$f_4 = (x_3 + x_4)(x_2 + x_4)(\bar{x}_1 + x_4)(x_1 + \bar{x}_2 + \bar{x}_4)$$

The first three sum terms are the same in both f_3 and f_4; they can be shared in a combined circuit. These terms require three OR gates and six inputs. In addition, one 2-input OR gate and one 4-input AND gate are needed for f_3, and one 3-input OR gate and one 4-input AND gate are needed for f_4. Thus the combined circuit comprises five OR gates, two AND gates, and 19 inputs, for a total cost of 26. This cost is slightly higher than the cost of the circuit derived in Example 2.17.

These examples show that the complexities of the best SOP or POS implementations of given functions may be quite different. For the functions in Figures 2.64 and 2.65, the SOP form gives better results. But if we are interested in implementing the complements of the four functions in these figures, then the POS form would be less costly.

Sophisticated CAD tools used to synthesize logic functions will automatically perform the types of optimizations illustrated in the preceding examples.

2.16 Concluding Remarks

In this chapter we introduced the concept of logic circuits. We showed that such circuits can be implemented using logic gates and that they can be described using a mathematical model called Boolean algebra. Because practical logic circuits are often large, it is important to have good CAD tools to help the designer. We introduced the VHDL hardware description language

that can be used to specify circuits for use in a CAD tool. We urge the reader to start using CAD software for the design of logic circuits as soon as possible.

This chapter has attempted to provide the reader with an understanding of some aspects of synthesis and optimization for logic functions. Now that the reader is comfortable with the fundamental concepts, we can examine digital circuits of a more sophisticated nature. The next chapter describes circuits that perform arithmetic operations, which are a key part of computers.

2.17 Examples of Solved Problems

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

Example 2.20

Problem: Determine if the following equation is valid

$$\bar{x}_1\bar{x}_3 + x_2x_3 + x_1\bar{x}_2 = \bar{x}_1x_2 + x_1x_3 + \bar{x}_2\bar{x}_3$$

Solution: The equation is valid if the expressions on the left- and right-hand sides represent the same function. To perform the comparison, we could construct a truth table for each side and see if the truth tables are the same. An algebraic approach is to derive a canonical sum-of-products form for each expression.

Using the fact that $x + \bar{x} = 1$ (theorem 8*b*), we can manipulate the left-hand side as follows:

$$\begin{aligned}
\text{LHS} &= \bar{x}_1\bar{x}_3 + x_2x_3 + x_1\bar{x}_2 \\
&= \bar{x}_1(x_2 + \bar{x}_2)\bar{x}_3 + (x_1 + \bar{x}_1)x_2x_3 + x_1\bar{x}_2(x_3 + \bar{x}_3) \\
&= \bar{x}_1x_2\bar{x}_3 + \bar{x}_1\bar{x}_2\bar{x}_3 + x_1x_2x_3 + \bar{x}_1x_2x_3 + x_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3
\end{aligned}$$

These product terms represent the minterms 2, 0, 7, 3, 5, and 4, respectively.

For the right-hand side, we have

$$\begin{aligned}\text{RHS} &= \bar{x}_1x_2 + x_1x_3 + \bar{x}_2\bar{x}_3 \\ &= \bar{x}_1x_2(x_3 + \bar{x}_3) + x_1(x_2 + \bar{x}_2)x_3 + (x_1 + \bar{x}_1)\bar{x}_2\bar{x}_3 \\ &= \bar{x}_1x_2x_3 + \bar{x}_1x_2\bar{x}_3 + x_1x_2x_3 + x_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2\bar{x}_3\end{aligned}$$

These product terms represent the minterms 3, 2, 7, 5, 4, and 0, respectively. Since both expressions specify the same minterms, they represent the same function; therefore, the equation is valid. Another way of representing this function is by $\sum m$ (0, 2, 3, 4, 5, 7).

Example 2.21

Problem: Design the minimum-cost product-of-sums expression for the function $f(x_1, x_2, x_3, x_4) = \sum$ m (0, 2, 4, 5, 6, 7, 8, 10, 12, 14, 15).

Solution: The function is defined in terms of its minterms. To find a POS expression we should start with the definition in terms of maxterms, which is $f = \Pi M$ (1, 3, 9, 11, 13). Thus,

$$f = M_1 \cdot M_3 \cdot M_9 \cdot M_{11} \cdot M_{13} = (x_1 + x_2 + x_3 + \bar{x}_4)(x_1 + x_2 + \bar{x}_3 + \bar{x}_4)(\bar{x}_1 + x_2 + x_3 + \bar{x}_4)$$
$$(\bar{x}_1 + x_2 + \bar{x}_3 + \bar{x}_4)(\bar{x}_1 + \bar{x}_2 + x_3 + \bar{x}_4)$$

We can rewrite the product of the first two maxterms as

$$\begin{aligned}M_1 \cdot M_3 &= (x_1 + x_2 + \bar{x}_4 + x_3)(x_1 + x_2 + \bar{x}_4 + \bar{x}_3) && \text{using commutative property } 10b \\ &= x_1 + x_2 + \bar{x}_4 && \text{using combining property } 14b\end{aligned}$$

Similarly, $M_9 \cdot M_{11} = \bar{x}_1 + x_2 + \bar{x}_4$. Now, we can use M_9 again, according to property $7a$, to derive $M_9 \cdot M_{13} = \bar{x}_1 + x_3 + \bar{x}_4$. Hence

$$f = (x_1 + x_2 + \bar{x}_4)(\bar{x}_1 + x_2 + \bar{x}_4)(\bar{x}_1 + x_3 + \bar{x}_4)$$

Applying $12b$ again, we get the final answer

$$f = (x_2 + \bar{x}_4)(\bar{x}_1 + x_3 + \bar{x}_4)$$

Example 2.22

Problem: A circuit that controls a given digital system has three inputs: x_1, x_2, and x_3. It has to recognize three different conditions:

- Condition A is true if x_3 is true and either x_1 is true or x_2 is false
- Condition B is true if x_1 is true and either x_2 or x_3 is false
- Condition C is true if x_2 is true and either x_1 is true or x_3 is false

The control circuit must produce an output of 1 if at least two of the conditions A, B, and C are true. Design the simplest circuit that can be used for this purpose.

Solution: Using 1 for true and 0 for false, we can express the three conditions as follows:

$$\begin{aligned} A &= x_3(x_1 + \bar{x}_2) = x_3x_1 + x_3\bar{x}_2 \\ B &= x_1(\bar{x}_2 + \bar{x}_3) = x_1\bar{x}_2 + x_1\bar{x}_3 \\ C &= x_2(x_1 + \bar{x}_3) = x_2x_1 + x_2\bar{x}_3 \end{aligned}$$

Then, the desired output of the circuit can be expressed as $f = AB + AC + BC$. These product terms can be determined as:

$$\begin{aligned} AB &= (x_3x_1 + x_3\bar{x}_2)(x_1\bar{x}_2 + x_1\bar{x}_3) \\ &= x_3x_1x_1\bar{x}_2 + x_3x_1x_1\bar{x}_3 + x_3\bar{x}_2x_1\bar{x}_2 + x_3\bar{x}_2x_1\bar{x}_3 \\ &= x_3x_1\bar{x}_2 + 0 + x_3\bar{x}_2x_1 + 0 \\ &= x_1\bar{x}_2x_3 \end{aligned}$$

$$\begin{aligned} AC &= (x_3x_1 + x_3\bar{x}_2)(x_2x_1 + x_2\bar{x}_3) \\ &= x_3x_1x_2x_1 + x_3x_1x_2\bar{x}_3 + x_3\bar{x}_2x_2x_1 + x_3\bar{x}_2x_2\bar{x}_3 \\ &= x_3x_1x_2 + 0 + 0 + 0 \\ &= x_1x_2x_3 \end{aligned}$$

$$\begin{aligned} BC &= (x_1\bar{x}_2 + x_1\bar{x}_3)(x_2x_1 + x_2\bar{x}_3) \\ &= x_1\bar{x}_2x_2x_1 + x_1\bar{x}_2x_2\bar{x}_3 + x_1\bar{x}_3x_2x_1 + x_1\bar{x}_3x_2\bar{x}_3 \\ &= 0 + 0 + x_1\bar{x}_3x_2 + x_1\bar{x}_3x_2 \\ &= x_1x_2\bar{x}_3 \end{aligned}$$

Therefore, f can be written as

$$\begin{aligned} f &= x_1\bar{x}_2x_3 + x_1x_2x_3 + x_1x_2\bar{x}_3 \\ &= x_1(\bar{x}_2 + x_2)x_3 + x_1x_2(x_3 + \bar{x}_3) \\ &= x_1x_3 + x_1x_2 \\ &= x_1(x_3 + x_2) \end{aligned}$$

Example 2.23

Problem: Solve the problem in Example 2.22 by using Venn diagrams.

Solution: The Venn diagrams for functions *A*, *B*, and *C* in Example 2.22 are shown in parts *a* to *c* of Figure 2.66. Since the function *f* has to be true when two or more of *A*, *B*, and *C* are true, then the Venn diagram for *f* is formed by identifying the common shaded areas in the Venn diagrams for *A*, *B*, and *C*. Any area that is shaded in two or more of these diagrams is also shaded in *f*, as shown in Figure 2.66*d*. This diagram corresponds to the function

$$f = x_1x_2 + x_1x_3 = x_1(x_2 + x_3)$$

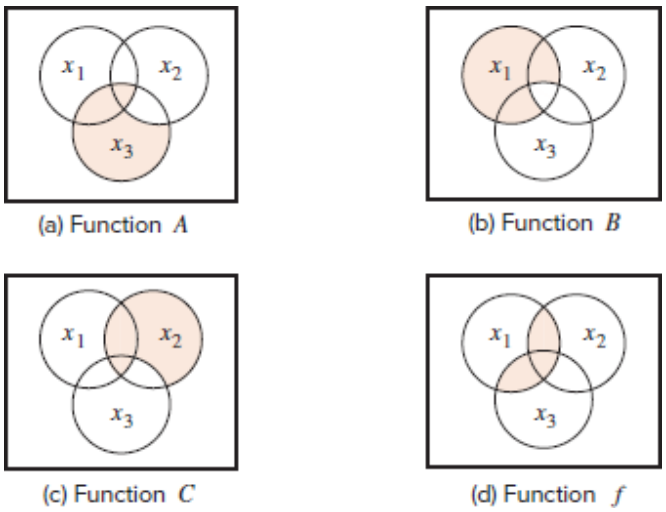


Figure 2.66 The Venn diagrams for Example 2.23.

Example 2.24

Problem: Use algebraic manipulation to derive the simplest sum-of-products expression for the function

$$f = x_2\bar{x}_3x_4 + x_1x_3x_4 + x_1\bar{x}_2x_4$$

Solution: Applying the consensus property 17*a* to the first two terms yields

$$\begin{aligned} f &= x_2\bar{x}_3x_4 + x_1x_3x_4 + x_2x_4x_1x_4 + x_1\bar{x}_2x_4 \\ &= x_2\bar{x}_3x_4 + x_1x_3x_4 + x_1x_2x_4 + x_1\bar{x}_2x_4 \end{aligned}$$

Now, using the combining property 14*a* for the last two terms gives

$$f = x_2\bar{x}_3x_4 + x_1x_3x_4 + x_1x_4$$

Finally, using the absorption property 13*a* produces

$$f = x_2\bar{x}_3x_4 + x_1x_4$$

Example 2.25

Problem: Use algebraic manipulation to derive the simplest product-of-sums expression for the function

$$f = (\bar{x}_1 + x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_4)(\bar{x}_1 + x_3 + x_4)$$

Solution: Applying the consensus property 17*b* to the first two terms yields

$$\begin{aligned} f &= (\bar{x}_1 + x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_4)(\bar{x}_1 + x_3 + \bar{x}_1 + \bar{x}_4)(\bar{x}_1 + x_3 + x_4) \\ &= (\bar{x}_1 + x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_4)(\bar{x}_1 + x_3 + \bar{x}_4)(\bar{x}_1 + x_3 + x_4) \end{aligned}$$

Now, using the combining property 14*b* for the last two terms gives

$$f = (\bar{x}_1 + x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_4)(\bar{x}_1 + x_3)$$

Finally, using the absorption property 13*b* on the first and last terms produces

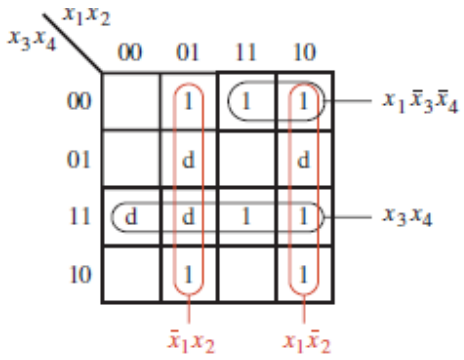
$$f = (\bar{x}_1 + \bar{x}_2 + \bar{x}_4)(\bar{x}_1 + x_3)$$

Example 2.26

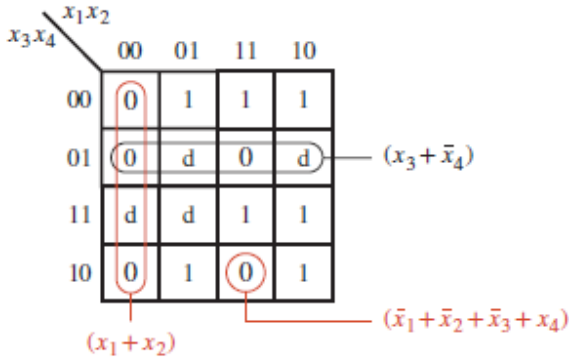
Problem: Determine the minimum-cost SOP and POS expressions for the function $f(x_1, x_2, x_3, x_4) = \sum m(4, 6, 8, 10, 11, 12, 15) + D(3, 5, 7, 9)$.

Solution: The function can be represented in the form of a Karnaugh map as shown in Figure 2.67*a*. Note that the location of minterms in the map is as indicated in Figure 2.53. To find the minimum-cost SOP expression, it is necessary to find the prime implicants that cover all 1s in the map. The don't-cares may be used as desired. Minterm m_6 is covered only by the prime

implicant $\bar{x}_1 x_2$; hence this prime implicant is essential and it must be included in the final expression. Similarly, the prime implicants $x_1\bar{x}_2$ and $x_3\ x_4$ are essential because they are the only ones that cover m_{10} and m_{15}, respectively. These three prime implicants cover all minterms for which f =1 except m_{12}. This minterm can be covered in two ways, by choosing either $x_1\bar{x}_3\bar{x}_4$ or $x_2\bar{x}_3\bar{x}_4$. Since both of these prime implicants have the same cost, we can choose either of them. Choosing the former, the desired SOP expression is



(a) Determination of the SOP expression



(b) Determination of the POS expression

Figure 2.67 Karnaugh maps for Example 2.26.

$$f = \bar{x}_1 x_2 + x_1\bar{x}_2 + x_3x_4 + x_1\bar{x}_3\bar{x}_4$$

These prime implicants are encircled in the map.

The desired POS expression can be found as indicated in Figure 2.67*b*. In this case, we have to find the sum terms that cover all 0s in the function. Note that we have written 0s explicitly in the map to emphasize this fact. The term (x_1 + x_2) is essential to cover the 0s in squares 0 and 2, which correspond to

maxterms M_0 and M_2. The terms $(x_3 + \bar{x}_4)$ and $(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + x_4)$ must be used to cover the 0s in squares 13 and 14, respectively. Since these three sum terms cover all 0s in the map, the POS expression is

$$f = (x_1 + x_2)(x_3 + \bar{x}_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + x_4)$$

The chosen sum terms are encircled in the map.

Observe the use of don't-cares in this example. To get a minimum-cost SOP expression we assumed that all four don't-cares have the value 1. But, the minimum-cost POS expression becomes possible only if we assume that don't-cares 3, 5, and 9 have the value 0 while the don't-care 7 has the value 1. This means that the resulting SOP and POS expressions are not identical in terms of the functions they represent. They cover identically all valuations for which *f* is specified as 1 or 0, but they differ in the valuations 3, 5, and 9. Of course, this difference does not matter, because the don't-care valuations will never be applied as inputs to the implemented circuits.

Example 2.27

Problem: Use Karnaugh maps to find the minimum-cost SOP and POS expressions for the function

$$f(x_1, \ldots, x_4) = \bar{x}_1\bar{x}_3\bar{x}_4 + x_3x_4 + \bar{x}_1\bar{x}_2x_4 + x_1x_2\bar{x}_3x_4$$

assuming that there are also don't-cares defined as $D = \sum (9, 12, 14)$.

Solution: The Karnaugh map that represents this function is shown in Figure 2.68*a*. The map is derived by placing 1s that correspond to each product term in the expression used to specify *f*. The term $\bar{x}_1\bar{x}_3\bar{x}_4$ corresponds to minterms 0 and 4. The term x_3 x_4 represents the third row in the map, comprising minterms 3, 7, 11, and 15. The term $\bar{x}_1\bar{x}_2x_4$ specifies minterms 1 and 3. The fourth product term represents the minterm 13. The map also includes the three don't-care conditions.

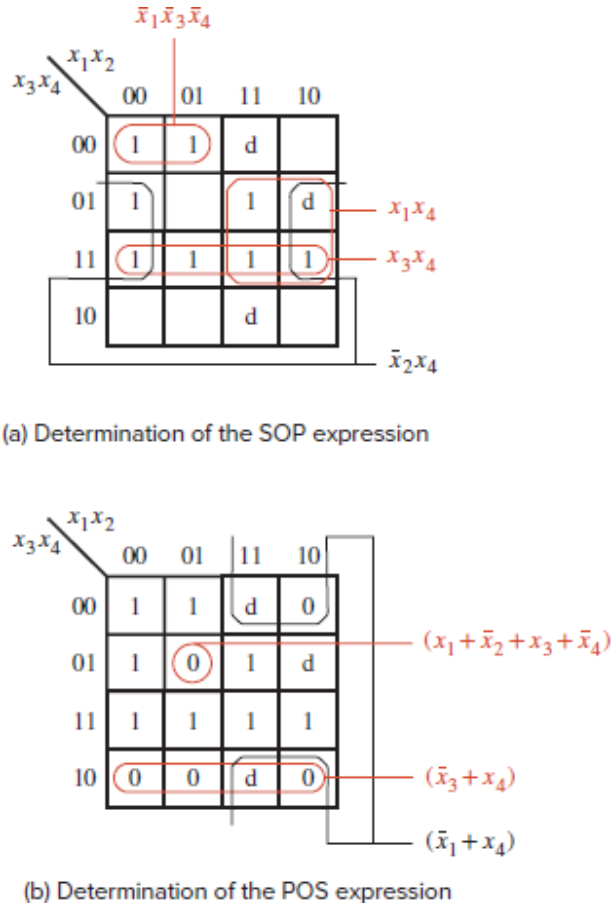


Figure 2.68 Karnaugh maps for Example 2.27.

To find the desired SOP expression, we must find the least-expensive set of prime implicants that covers all 1s in the map. The term $x_3\ x_4$ is a prime implicant which must be included because it is the only prime implicant that covers the minterm 7; it also covers minterms 3, 11, and 15. Minterm 4 can be covered with either $\bar{x}_1\bar{x}_3\bar{x}_4$ or $x_2\bar{x}_3\bar{x}_4$. Both of these terms have the same cost; we will choose $\bar{x}_1\bar{x}_3\bar{x}_4$ because it also covers the minterm 0. Minterm 1 may be covered with either $\bar{x}_1\bar{x}_2\bar{x}_3$ or $\bar{x}_2x_4$; we should choose the latter because its cost is lower. This leaves only the minterm 13 to be covered, which can be done with either $x_1\ x_4$ or $x_1\ x_2$ at equal costs. Choosing $x_1\ x_4$, the minimum-cost SOP expression is

$$f = x_3x_4 + \bar{x}_1\bar{x}_3\bar{x}_4 + \bar{x}_2x_4 + x_1x_4$$

Figure 2.68*b* shows how we can find the POS expression. The sum term $(\bar{x}_3 + x_4)$ covers the 0s in the bottom row. To cover the 0 in square 8 we must

include $(\bar{x}_1 + x_4)$. The remaining 0, in square 5, must be covered with $(x_1 + \bar{x}_2 + x_3 + \bar{x}_4)$. Thus, the minimum-cost POS expression is

$$f = (\bar{x}_3 + x_4)(\bar{x}_1 + x_4)(x_1 + \bar{x}_2 + x_3 + \bar{x}_4)$$

Example 2.28

Problem: Consider the expression

$$f = s_3(\bar{s}_1 + \bar{s}_2) + s_1 s_2$$

Derive a minimum-cost SOP expression for *f*.
Solution: Applying the distributive property 12*a* we can write

$$f = \bar{s}_1 s_3 + \bar{s}_2 s_3 + s_1 s_2$$

Now, it is easy to see how *f* can be represented in the form of a Karnaugh map, as depicted in Figure 2.69. The figure shows that minterms from each of the above three product terms cover the bottom row of the Karnaugh map, leading to the minimum-cost expression

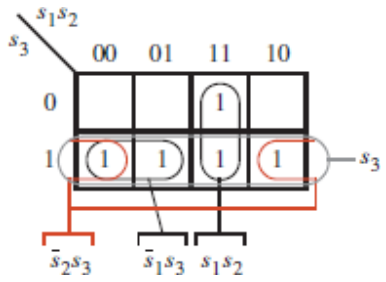


Figure 2.69 A Karnaugh map that represents the function in Example 2.28.

$$f = s_3 + s_1 s_2$$

Example 2.29

write VHDL code that represents the logic circuit in Figure 2.70. Use only signal assignment statements to specify the required functions.

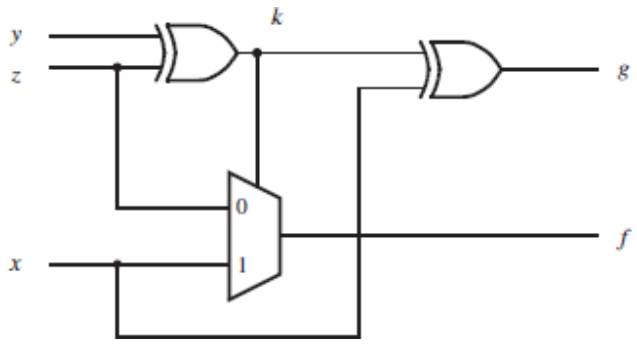


Figure 2.70 The logic circuit for Example 2.29.

Solution: An example of the required VHDL code is shown in Figure 2.71.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY f_g IS
    PORT ( x, y, z  : IN   STD_LOGIC ;
           f, g     : OUT  STD_LOGIC ) ;
END f_g ;

ARCHITECTURE LogicFunc OF f_g IS
    SIGNAL k : STD_LOGIC ;
BEGIN
    k <= y XOR z ;
    g <= k XOR x ;
    f <= (NOT k AND z) OR (k AND x) ;
END LogicFunc ;
```

Figure 2.71 VHDL code for Example 2.29.

Example 2.30

consider the circuit shown in Figure 2.72. It includes two copies of the 2-to-1 multiplexer circuit from Figure 2.33, and the adder circuit from Figure 2.12. If the multiplexers' select input m=0, then this circuit produces the sum S = a + b. But if m=1, then the circuit produces S = c + d. By using multiplexers, we are able to *share* one adder for generating the two different sums $a + b$ and $c + d$. Sharing a subcircuit for multiple purposes is commonly used in practice, although usually with larger subcircuits than our one-bit adder.

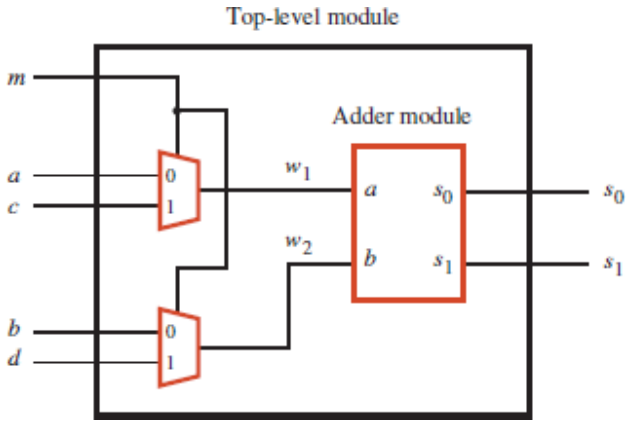


Figure 2.72 The circuit for Example 2.30.

Write VHDL code for the circuit in Figure 2.72. Use the hierarchical style of VHDL code illustrated in Figure 2.47, including two instances of a VHDL entity for the 2-to-1 multiplexer subcircuit and one instance of the adder subcircuit.

Solution: An example of the required VHDL code is shown in Figure 2.73. The VHDL code for the *example3* entity, which represents the 2-to-1 multiplexer subcircuit, is shown in Figure 2.42, and the *adder* entity appears in Figure 2.45.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY example2_30 IS
    PORT ( a, b, c, d, m   : IN     STD_LOGIC ;
           s1, s0          : OUT  STD_LOGIC ) ;
END example2_30 ;

ARCHITECTURE Structure OF example2_30 IS
    COMPONENT example3 IS
        PORT ( x1, x2, s   : IN     STD_LOGIC ;
               f           : OUT  STD_LOGIC ) ;
    END COMPONENT ;
    COMPONENT adder IS
        PORT ( a, b        : IN     STD_LOGIC ;
               s1, s0      : OUT  STD_LOGIC ) ;
    END COMPONENT ;
    SIGNAL w1, w2 : STD_LOGIC ;
BEGIN
    U1: example3 PORT MAP ( a, c, m, w1 ) ;
    U2: example3 PORT MAP ( b, d, m, w2 ) ;
    U3: adder PORT MAP ( w1, w2, s1, s0 ) ;
END Structure ;
```

Figure 2.73 VHDL code for Example 2.30.

Example 2.31

In Chapter 1 we said that several types of integrated circuit chips are available for implementation of logic circuits, and that field-programmable gate arrays (FPGAs) are commonly used. In an FPGA, logic functions are not implemented directly using AND, OR, and NOT gates. Instead, an FPGA implements logic functions by using a type of circuit element called lookup tables (LUTs). A LUT can be programmed by the user to implement any logic function of its inputs. Thus, if a LUT has three inputs, then it can implement any logic function of these three inputs. We describe FPGAs and lookup tables in detail in Appendix B. Consider the four-input function

$$f = x_1x_2x_4 + x_2x_3\overline{x}_4 + \overline{x}_1\overline{x}_2\overline{x}_3$$

Show how this function can be implemented in an FPGA that has three-input LUTs.

Solution: A straightforward implementation requires four 3-input LUTs. Three of these LUTs are used for the three-input AND operations, and the final LUT implements the three-input OR operation. It is also possible to apply logic synthesis techniques that result in fewer LUTs. We discuss these techniques in Chapter 8 (see Example 8.19).

Problems

Answers to problems marked by an asterisk are given at the back of the book.

2.1 Use algebraic manipulation to prove that $x + yz = (x + y) \cdot (x + z)$. Note that this is the distributive rule, as stated in identity 12*b* in Section 2.5.

2.2 Use algebraic manipulation to prove that $(x + y) \cdot (x + \overline{y}) = x$.

2.3 Use algebraic manipulation to prove that $xy + yz + \overline{x}z = xy + \overline{x}z$. Note that this is the consensus property 17*a* in Section 2.5.

2.4 Use the Venn diagram to prove the identity in Problem 2.3.

2.5 Use the Venn diagram to prove DeMorgan's theorem, as given in expression 15*b* in Section 2.5.

2.6 Use the Venn diagram to prove that

$$(x_1 + x_2 + x_3) \cdot (x_1 + x_2 + \bar{x}_3) = x_1 + x_2$$

***2.7** Determine whether or not the following expressions are valid, i.e., whether the left- and right-hand sides represent the same function

(a) $\bar{x}_1 x_3 + x_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 + x_1 \bar{x}_2 = \bar{x}_2 x_3 + x_1 \bar{x}_3 + x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3$

(b) $x_1 \bar{x}_3 + x_2 x_3 + \bar{x}_2 \bar{x}_3 = (x_1 + \bar{x}_2 + x_3)(x_1 + x_2 + \bar{x}_3)(\bar{x}_1 + x_2 + \bar{x}_3)$

(c) $(x_1 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + x_2) = (x_1 + x_2)(x_2 + x_3)(\bar{x}_1 + \bar{x}_3)$

2.8 Draw a timing diagram for the circuit in Figure 2.24*a*. Show the waveforms that can be observed on all wires in the circuit.

2.9 Repeat Problem 2.8 for the circuit in Figure 2.24*b*.

2.10 Use algebraic manipulation to show that for three input variables x_1, x_2, and x_3

$$\sum m(1, 2, 3, 4, 5, 6, 7) = x_1 + x_2 + x_3$$

2.11 Use algebraic manipulation to show that for three input variables x_1, x_2, and x_3

$$\Pi M(0, 1, 2, 3, 4, 5, 6) = x_1 x_2 x_3$$

***2.12** Use algebraic manipulation to find the minimum sum-of-products expression for the function $f = x_1 x_3 + x_1 \bar{x}_2 + \bar{x}_1 x_2 x_3 + \bar{x}_1 \bar{x}_2 \bar{x}_3$.

2.13 Use algebraic manipulation to find the minimum sum-of-products expression for the function $f = x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 x_4 + x_1 \bar{x}_2 x_3 \bar{x}_4$.

2.14 Use algebraic manipulation to find the minimum product-of-sums expression for the function $f = (x_1 + x_3 + x_4) \cdot (x_1 + \bar{x}_2 + x_3) \cdot (x_1 + \bar{x}_2 + \bar{x}_3 + x_4)$.

***2.15** Use algebraic manipulation to find the minimum product-of-sums expression for the function

$f = (x_1 + x_2 + x_3) \cdot (x_1 + \bar{x}_2 + x_3) \cdot (\bar{x}_1 + \bar{x}_2 + x_3) \cdot (x_1 + x_2 + \bar{x}_3)$.

2.16 (a) Show the location of all minterms in a three-variable Venn diagram.

(b) Show a separate Venn diagram for each product term in the function $f = x_1 \bar{x}_2 x_3 + x_1 x_2 + \bar{x}_1 x_3$.

2.17 Represent the function in Figure 2.23 in the form of a Venn diagram and find its minimal sum-of-products form.

2.18 Figure P2.1 shows two attempts to draw a Venn diagram for four variables. For parts (*a*) and (*b*) of the figure, explain why the Venn diagram is not correct. (*Hint:* The Venn diagram must be able to represent all 16 minterms of the four variables.)

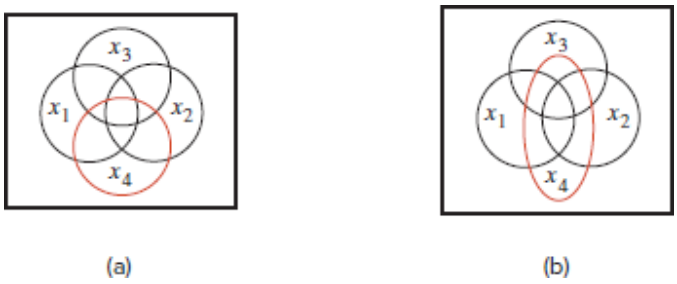


Figure P2.1 Two attempts to draw a four-variable Venn diagram.

2.19 Figure P2.2 gives a representation of a four-variable Venn diagram and shows the location of minterms m_0, m_1, and m_2. Show the location of the other minterms in the diagram. Represent the function $f = \bar{x}_1\bar{x}_2x_3\bar{x}_4 + x_1x_2x_3x_4 + \bar{x}_1x_2$ on this diagram.

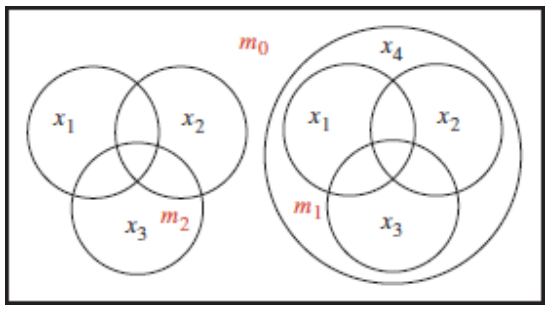


Figure P2.1 A four-variable Venn diagram.

***2.20** Design the simplest sum-of-products circuit that implements the function $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$.

2.21 Design the simplest sum-of-products circuit that implements the function $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$.

2.22 Design the simplest product-of-sums circuit that implements the function $f(x_1, x_2, x_3) = \Pi M(0, 2, 5)$.

***2.23** Design the simplest product-of-sums expression for the function $f(x_1, x_2, x_3) = \Pi M(0, 1, 5, 7)$.

2.24 Derive the simplest sum-of-products expression for the function $f(x_1, x_2, x_3, x_4) = x_1\bar{x}_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_1\bar{x}_2\bar{x}_3$.

2.25 Use algebraic manipulation to derive the simplest sum-of-products expression for the function $f(x_1, x_2, x_3, x_4, x_5) = \bar{x}_1\bar{x}_3\bar{x}_5 + \bar{x}_1\bar{x}_3\bar{x}_4 + \bar{x}_1x_4x_5 + x_1\bar{x}_2\bar{x}_3x_5$ (*Hint:* Use the consensus property 17*a*.)

2.26 Use algebraic manipulation to derive the simplest product-of-sums expression for the function $f(x_1, x_2, x_3, x_4) = (\bar{x}_1 + \bar{x}_3 + \bar{x}_4)(\bar{x}_2 + \bar{x}_3 + x_4)(x_1 + \bar{x}_2 + \bar{x}_3)$ (*Hint:* Use the consensus property 17*b*.)

2.27 Use algebraic manipulation to derive the simplest product-of-sums expression for the function $f(x_1, x_2, x_3, x_4, x_5) = (\bar{x}_2 + x_3 + x_5)(x_1 + \bar{x}_3 + x_5)(x_1 + x_2 + x_5)(x_1 + \bar{x}_4 + \bar{x}_5)$ (*Hint:* Use the consensus property 17*b*.)

***2.28** Design the simplest circuit that has three inputs, x_1, x_2, and x_3, which produces an output value of 1 whenever two or more of the input variables have the value 1; otherwise, the output has to be 0.

2.29 Design the simplest circuit that has three inputs, x_1, x_2, and x_3, which produces an output value of 1 whenever exactly one or two of the input variables have the value 1; otherwise, the output has to be 0.

2.30 Design the simplest circuit that has four inputs, x_1, x_2, x_3, and x_4, which produces an output value of 1 whenever three or more of the input variables have the value 1; otherwise, the output has to be 0.

2.31 For the timing diagram in Figure P2.3, synthesize the function $f(x_1, x_2, x_3)$ in the simplest sum-of-products form.

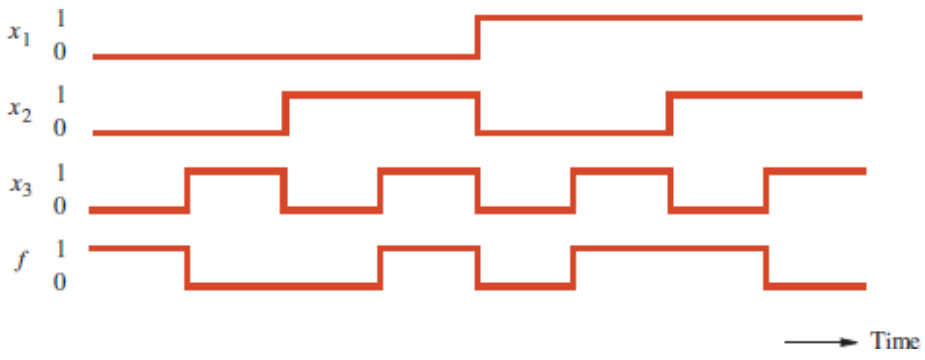


Figure P2.3 A timing diagram representing a logic function.

***2.32** For the timing diagram in Figure P2.3, synthesize the function $f(x_1, x_2, x_3)$ in the simplest product-of-sums form.

***2.33** For the timing diagram in Figure P2.4, synthesize the function $f(x_1, x_2, x_3)$ in the simplest sum-of-products form.

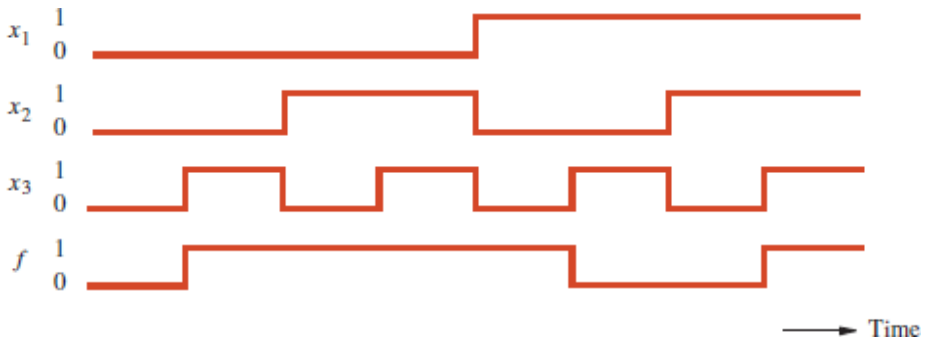


Figure P2.4 A timing diagram representing a logic function.

2.34 For the timing diagram in Figure P2.4, synthesize the function $f(x_1, x_2, x_3)$ in the simplest product-of-sums form.

2.35 Design a circuit with output f and inputs x_1, x_0, y_1, and y_0. Let $X = x_1\ x_0$ and $Y = y_1\ y_0$ represent two 2-digit binary numbers. The output f should be 1 if the numbers represented by X and Y are equal. Otherwise, f should be 0 (a) Show the truth table for f (b) Synthesize the simplest possible product-of-sums expression for f.

2.36 Repeat Problem 2.35 for the case where f should be 1 only if $X \geq Y$ (a) Show the truth table for f (b) Show the canonical sum-of-products expression for f (c) Show the simplest possible sum-of-products expression for f.

***2.37** Find the minimum-cost SOP and POS forms for the function $f(x_1, x_2, x_3) = \sum m(1,2,3,5)$.

***2.38** Repeat Problem 2.37 for the function $f(x_1, x_2, x_3) = \sum m(1,4,7) + D(2,5)$.

2.39 Repeat Problem 2.37 for the function

$$f(x_1, \ldots, x_4) = \Pi M(0, 1, 2, 4, 5, 7, 8, 9, 10, 12, 14, 15).$$

2.40 Repeat Problem 2.37 for the function

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 8, 9, 10, 15) + D(1, 3, 6, 7).$$

***2.41** Repeat Problem 2.37 for the function

$$f(x_1, \ldots, x_5) = \Pi M(1, 4, 6, 7, 9, 12, 15, 17, 20, 21, 22, 23, 28, 31).$$

2.42 Repeat Problem 2.37 for the function

$$f(x_1, \ldots, x_5) = \sum m(0, 1, 3, 4, 6, 8, 9, 11, 13, 14, 16, 19, 20, 21, 22, 24, 25) + D(5, 7, 12, 15, 17, 23)$$

.

2.43 Repeat Problem 2.37 for the function

$$f(x_1, \ldots, x_5) = \sum m(1, 4, 6, 7, 9, 10, 12, 15, 17, 19, 20, 23, 25, 26, 27, 28, 30, 31) + D(8, 16, 21, 22)$$

.

2.44 Find 5 three-variable functions for which the product-of-sums form has lower cost than the sum-of-products form.

***2.45** A four-variable logic function that is equal to 1 if any three or all four of its variables are equal to 1 is called a *majority* function. Design a minimum-cost SOP circuit that implements this majority function.

2.46 Derive a minimum-cost realization of the four-variable function that is equal to 1 if exactly two or exactly three of its variables are equal to 1; otherwise it is equal to 0.

***2.47** Prove or show a counter-example for the statement: If a function f has a unique minimum-cost SOP expression, then it also has a unique minimum-cost POS expression.

***2.48** A circuit with two outputs has to implement the following functions:

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 6, 7, 9) + D(10, 11)$$
$$g(x_1, \ldots, x_4) = \sum m(2, 4, 9, 10, 15) + D(0, 13, 14)$$

Design the minimum-cost circuit and compare its cost with combined costs of two circuits that implement f and g separately. Assume that the input variables are available in both uncomplemented and complemented forms.

2.49 Repeat Problem 2.48 for the following functions:

$$f(x_1, \ldots, x_5) = \sum m(1, 4, 5, 11, 27, 28) + D(10, 12, 14, 15, 20, 31)$$
$$g(x_1, \ldots, x_5) = \sum m(0, 1, 2, 4, 5, 8, 14, 15, 16, 18, 20, 24, 26, 28, 31) + D(10, 11, 12, 27)$$

***2.50** Consider the function $f = x_3x_5 + \bar{x}_1x_2x_4 + x_1\bar{x}_2\bar{x}_4 + x_1x_3\bar{x}_4 + \bar{x}_1x_3x_4 + \bar{x}_1x_2x_5 + x_1\bar{x}_2x_5$. Derive a minimum-cost POS expression for this function.

2.51 Implement the function in Figure 2.31 using only NAND gates.

2.52 Implement the function in Figure 2.31 using only NOR gates.

2.53 Implement the circuit in Figure 2.41 using NAND and NOR gates.

***2.54** Design the simplest circuit that implements the function $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$ using NAND gates.

2.55 Design the simplest circuit that implements the function $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$ using NAND gates.

***2.56** Repeat Problem 2.54 using NOR gates.

2.57 Repeat Problem 2.55 using NOR gates.

2.58 (a) Use a schematic capture tool (which can be downloaded from the Internet; for example, from the websites for Intel or Xilinx) to draw

schematics for the following functions:

$$f_1 = x_2\bar{x}_3\bar{x}_4 + \bar{x}_1x_2x_4 + \bar{x}_1x_2x_3 + x_1x_2x_3$$
$$f_2 = x_2\bar{x}_4 + \bar{x}_1x_2 + x_2x_3$$

(b) Use functional simulation to prove that $f_1 = f_2$.

2.59 (a) Use a schematic capture tool to draw schematics for the following functions:

$$f_1 = (x_1 + x_2 + \bar{x}_4) \cdot (\bar{x}_2 + x_3 + \bar{x}_4) \cdot (\bar{x}_1 + x_3 + \bar{x}_4) \cdot (\bar{x}_1 + \bar{x}_3 + \bar{x}_4)$$
$$f_2 = (x_2 + \bar{x}_4) \cdot (x_3 + \bar{x}_4) \cdot (\bar{x}_1 + \bar{x}_4)$$

(b) Use functional simulation to prove that $f_1 = f_2$.

2.60 Write VHDL code to implement the circuit in Figure 2.32*a* using Boolean operators.

2.61 Repeat Problem 2.60 for the circuit in Figure 2.32*b*.

2.62 Write VHDL code to implement the function $f(x_1, x_2, x_3) = \sum m(1, 2, 3, 4, 5, 6)$ using Boolean operators. Ensure that the resulting circuit is as simple as possible.

2.63 Write VHDL code to implement the function $f(x_1, x_2, x_3) = \sum m(0, 1, 3, 4, 5, 6)$ using signal assignment.

2.64 (a) Write VHDL code to describe the following functions:

$$f_1 = x_1\bar{x}_3 + x_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2 + x_1\bar{x}_4$$
$$f_2 = (x_1 + \bar{x}_3) \cdot (x_1 + x_2 + \bar{x}_4) \cdot (x_2 + \bar{x}_3 + \bar{x}_4)$$

(b) Use functional simulation to prove that $f_1 = f_2$.

2.65 Consider the following VHDL statements:

```
f1 <= (x1 AND x3) OR (NOT x1 AND NOT x3) OR (x2 AND x4) OR
      (NOT x2 AND NOT x4) ;
f2 <= (x1 AND x2 AND NOT x3 AND NOT x4) OR
      (NOT x1 AND NOT x2 AND x3 AND x4) OR
      (x1 AND NOT x2 AND NOT x3 AND x4) OR
      (NOT x1 AND x2 AND x3 AND NOT x4).
```

(a) Write complete VHDL code to implement f1 and f2.

(b) Use functional simulation to prove that $f1 = \overline{f2}$.

***2.66** Consider the logic expressions:

$$f = x_1\bar{x}_2\bar{x}_5 + \bar{x}_1\bar{x}_2\bar{x}_4\bar{x}_5 + x_1x_2x_4x_5 + \bar{x}_1\bar{x}_2x_3\bar{x}_4 + x_1\bar{x}_2x_3x_5 + \bar{x}_2\bar{x}_3x_4\bar{x}_5 + x_1x_2x_3x_4\bar{x}_5$$
$$g = \bar{x}_2x_3\bar{x}_4 + \bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5 + x_1x_3x_4\bar{x}_5 + x_1\bar{x}_2x_4\bar{x}_5 + x_1x_3x_4x_5 + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_5 + x_1x_2\bar{x}_3x_4x_5$$

Prove or disprove that $f = g$.

2.67 Repeat Problem 2.66 for the following expressions:

$$f = x_1\bar{x}_2\bar{x}_3 + x_2x_4 + x_1\bar{x}_2\bar{x}_4 + \bar{x}_2\bar{x}_3\bar{x}_4 + \bar{x}_1x_2x_3$$
$$g = (\bar{x}_2 + x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + x_4)(x_2 + \bar{x}_3 + \bar{x}_4)(x_1 + x_2 + \bar{x}_3)(x_1 + x_2 + \bar{x}_4)$$

2.68 Repeat Problem 2.66 for the following expressions:

$$f = x_2\bar{x}_3\bar{x}_4 + \bar{x}_2x_3 + \bar{x}_2x_4 + x_1x_2\bar{x}_4 + x_1x_2\bar{x}_3\bar{x}_5$$
$$g = (x_2 + x_3 + x_4)(\bar{x}_2 + \bar{x}_4 + x_5)(x_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_2 + x_3 + \bar{x}_4 + \bar{x}_5)$$

2.69 A circuit with two outputs is defined by the logic functions:

$$f = x_1\bar{x}_2\bar{x}_3 + \bar{x}_2\bar{x}_4 + \bar{x}_2\bar{x}_3x_4 + x_1x_2x_3x_4$$
$$g = x_1\bar{x}_3x_4 + x_1x_2x_4 + \bar{x}_1x_3x_4 + \bar{x}_2x_3\bar{x}_4$$

Derive a minimum-cost implementation of this circuit. What is the cost of your circuit?

2.70 Repeat Problem 2.69 for the functions:

$$f = (\bar{x}_1 + x_2 + \bar{x}_3)(x_1 + x_3 + \bar{x}_4)(x_1 + \bar{x}_2 + x_3)(\bar{x}_1 + x_2 + x_4)(x_1 + \bar{x}_2 + \bar{x}_4)$$
$$g = (\bar{x}_1 + x_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_4)(\bar{x}_2 + \bar{x}_3 + \bar{x}_4)(x_1 + \bar{x}_2 + x_3 + x_4)$$

2.71 A given system has four sensors that can produce an output of 0 or 1. The system operates properly when exactly one of the sensors has its output equal to 1. An alarm must be raised when two or more sensors have the output of 1. Design the simplest circuit that can be used to raise the alarm.

2.72 Repeat Problem 2.71 for a system that has seven sensors.

2.73 Find the minimum-cost circuit consisting only of two-input NAND gates for the function $f(x_1, \ldots, x_4) = \sum m(0, 1, 2, 3, 4, 6, 8, 9, 12)$. Assume that the input variables are available in both uncomplemented and complemented forms. (*Hint:* Consider the complement of the function.)

2.74 Repeat Problem 2.73 for the function $f(x_1, \ldots, x_4) = \sum m(2, 3, 6, 8, 9, 12)$.

2.75 Find the minimum-cost circuit consisting only of two-input NOR gates for the function $f(x_1, \ldots, x_4) = \sum m(6, 7, 8, 10, 12, 14, 15)$. Assume that the input variables are available in both uncomplemented and

complemented forms. (*Hint:* Consider the complement of the function.)

2.76 Repeat Problem 2.75 for the function

$f(x_1, \ldots, x_4) = \sum m(2, 3, 4, 5, 9, 10, 11, 12, 13, 15)$.

2.77 Consider the circuit in Figure P2.5, which implements functions f and g. What is the cost of this circuit, assuming that the input variables are available in both true and complemented forms? Redesign the circuit to implement the same functions, but at as low a cost as possible. What is the cost of your circuit?

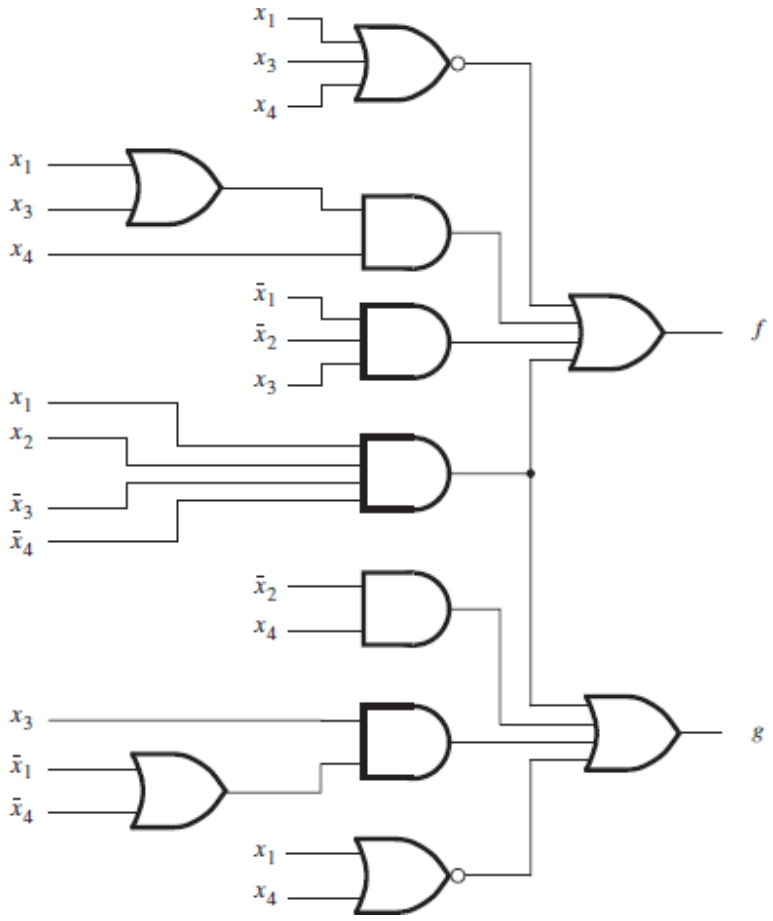


Figure P2.5 Circuit for Problem 2.77.

2.78 Repeat Problem 2.77 for the circuit in Figure P2.6. Use only NAND gates in your circuit.

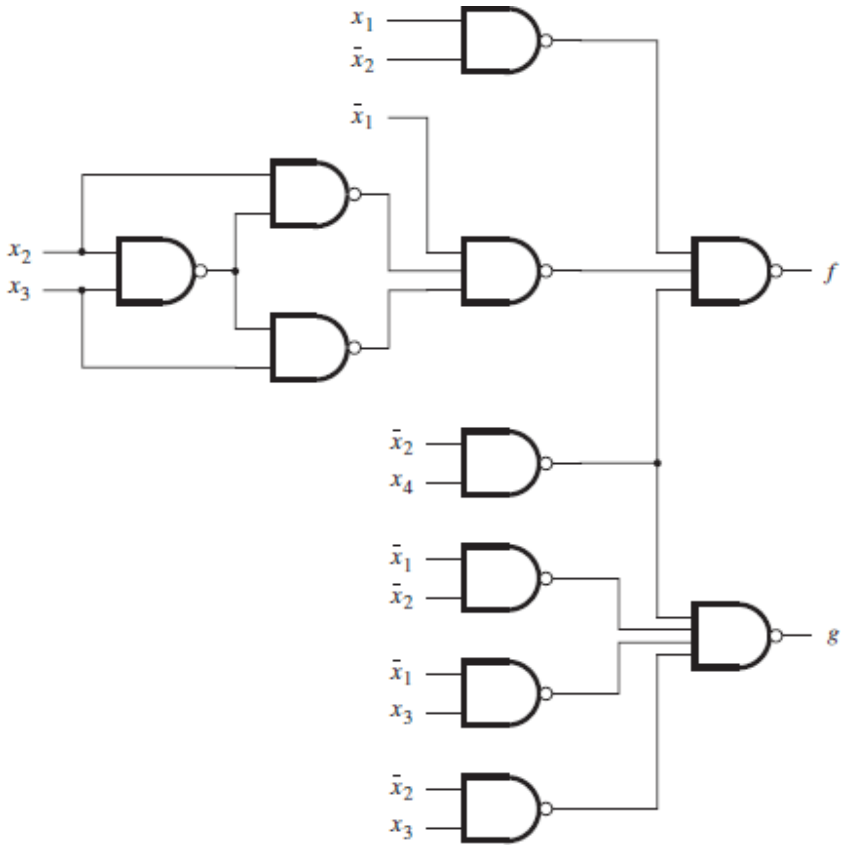


Figure P2.6 Circuit for Problem 2.78.

REFERENCES

1. G. Boole, *An Investigation of the Laws of Thought*, 1854, reprinted by Dover Publications, New York, 1954.
2. C. E. Shannon, “A Symbolic Analysis of Relay and Switching Circuits," *Transactions of AIEE* 57 (1938), pp. 713–723.
3. E. V. Huntington, “Sets of Independent Postulates for the Algebra of Logic,” *Transactions of the American Mathematical Society* 5 (1904), pp. 288–309.

chapter

3

NUMBER REPRESENTATION AND ARITHMETIC CIRCUITS

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Representation of numbers in computers
- Circuits used to perform arithmetic operations
- Performance issues in large circuits
- Use of VHDL to specify arithmetic circuits

In this chapter we will discuss logic circuits that perform arithmetic operations. We will explain how numbers can be added, subtracted, and multiplied. We will also show how to write VHDL code to describe the arithmetic circuits. These circuits provide an excellent platform for illustrating the power and versatility of VHDL in specifying complex logic-circuit assemblies. The concepts involved in the design of arithmetic circuits are easily applied to a wide variety of other circuits.

Before tackling the design of arithmetic circuits, it is necessary to discuss how numbers are represented in digital systems. In Chapter 1 we introduced binary numbers and showed how they can be expressed using the positional number representation. We also discussed the conversion process between decimal and binary number systems. In Chapter 2 we dealt with logic variables in a general way, using variables to represent either the states of switches or some general conditions. Now we will use the variables to represent numbers. Several variables are needed to specify a number, with each variable corresponding to one digit of the number.

3.1 Positional Number Representation

When dealing with numbers and arithmetic operations, it is convenient to use standard symbols. Thus to represent addition we use the plus (+) symbol, and for subtraction we use the minus (-) symbol. In Chapter 2 we used the + symbol mostly to represent the logical OR operation. Even though we will now use the same symbols for two different purposes, the meaning of each symbol will usually be clear from the context of the discussion. In cases where there may be some ambiguity, the meaning will be stated explicitly.

3.1.1 Unsigned Integers

The simplest numbers to consider are the integers. We will begin by considering positive integers and then expand the discussion to include negative integers. Numbers that are positive only are called *unsigned*, and numbers that can also be negative are called *signed*. Representation of numbers that include a radix point (real numbers) is discussed later in the chapter.

As explained in Section 1.5.1, an *n*-bit unsigned number

$$B = b_{n-1}b_{n-2}\cdots b_1b_0$$

represents an integer that has the value

$$V(B) = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \quad [3.1]$$
$$= \sum_{i=0}^{n-1} b_i \times 2^i$$

3.1.2 OCTAL AND HEXADECIMAL REPRESENTATIONS

The positional number representation can be used for any radix. If the radix is *r*, then the number

$$K = k_{n-1}k_{n-2}\cdots k_1k_0$$

has the value

$$V(K) = \sum_{i=0}^{n-1} k_i \times r^i$$

Our interest is limited to those radices that are most practical. We will use decimal numbers because they are used by people, and we will use binary numbers because they are used by computers. In addition, two other radices are useful—8 and 16. Numbers represented with radix 8 are called *octal* numbers, while radix-16 numbers are called *hexadecimal* numbers. In octal representation the digit values range from 0 to 7. In hexadecimal representation (often abbreviated as *hex*), each digit can have one of 16 values. The first ten are denoted the same as in the decimal system, namely, 0 to 9. Digits that correspond to the decimal values 10, 11, 12, 13, 14, and 15 are denoted by the letters, A, B, C, D, E, and F. Table 3.1 gives the first 18 integers in these number systems.

Table 3-1 Numbers in different systems.

Decimal	Binary	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

In computers the dominant number system is binary. The reason for using the octal and hexadecimal systems is that they serve as a useful shorthand notation for binary numbers. One octal digit represents three bits. Thus a binary number is converted into an octal number by taking groups of three bits, starting from the least-significant bit, and replacing them with the corresponding octal digit. For example, 101011010111 is converted as

$$\underbrace{101}_{5}\quad\underbrace{011}_{3}\quad\underbrace{010}_{2}\quad\underbrace{111}_{7}$$

which means that $(101011010111)_2 = (5327)_8$. If the number of bits is not a multiple of three, then we add 0s to the left of the most-significant bit. For example, $(10111011)_2 = (273)_8$ because of the grouping

$$\underbrace{010}_{2}\quad\underbrace{111}_{7}\quad\underbrace{011}_{3}$$

Conversion from octal to binary is just as straightforward; each octal digit is simply replaced by three bits that denote the same value.

Similarly, a hexadecimal digit represents four bits. For example, a 16-bit number is represented by four hex digits, as in

$$(1010111100100101)_2 = (AF25)_{16}$$

using the grouping

$$\underbrace{1010}_{A} \quad \underbrace{1111}_{F} \quad \underbrace{0010}_{2} \quad \underbrace{0101}_{5}$$

Zeros are added to the left of the most-significant bit if the number of bits is not a multiple of four. For example, $(1101101000)_2 = (368)_{16}$ because of the grouping

$$\underbrace{0011}_{3} \quad \underbrace{0110}_{6} \quad \underbrace{1000}_{8}$$

Conversion from hexadecimal to binary involves straightforward substitution of each hex digit by four bits that denote the same value.

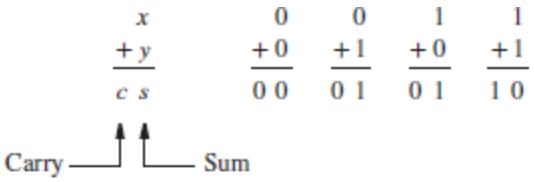
Binary numbers used in modern computers often have 32 or 64 bits. Written as binary *n*-tuples (sometimes called bit vectors), such numbers are awkward for people to deal with. It is much simpler to deal with them in the form of 8- or 16-digit hex numbers. Because the arithmetic operations in a digital system usually involve binary numbers, we will focus on circuits that use such numbers. We will sometimes use the hexadecimal representation as a convenient shorthand description.

We have introduced the simplest numbers—unsigned integers. It is necessary to be able to deal with several other types of numbers. We will discuss the representation of signed numbers, fixed-point numbers, and floating-point numbers later in this chapter. But first we will examine some simple circuits that operate on numbers to give the reader a feeling for digital circuits that perform arithmetic operations and to provide motivation for further discussion.

3.2 Addition of Unsigned Numbers

Binary addition is performed in the same way as decimal addition except that the values of individual digits can be only 0 or 1. In Chapter 2, we already considered the addition of 2 one-bit numbers, as an example of a simple logic circuit. Now, we will consider this task in the context of general adder circuits. The one-bit addition entails four possible

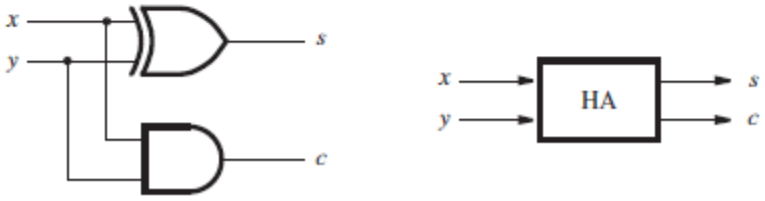
combinations, as indicated in Figure 3.1*a*. Two bits are needed to represent the result of the addition. The rightmost bit is called the *sum*, *s*. The leftmost bit, which is produced as a carry-out when both bits being added are equal to 1, is called the *carry*, *c*. The addition operation is defined in the form of a truth table in part (*b*) of the figure. The sum bit *s* is the XOR function. The carry *c* is the AND function of inputs *x* and *y*. A circuit realization of these functions is shown in Figure 3.1*c*. This circuit, which implements the addition of only two bits, is called a *half-adder*.



(a) The four possible cases

x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table



(c) Circuit (d) Graphical symbol

Figure 3.1 Half-adder.

A more interesting case is when larger numbers that have multiple bits are involved. Then it is still necessary to add each pair of bits, but for each bit position *i*, the addition operation may include a *carry-in* from bit position *i*-1.

Figure 3.2*a* presents an example of the addition operation. The two operands are $X = (01111)_2 = (15)_{10}$ and $Y = (01010)_2 = (10)_{10}$. Five bits are

used to represent X and Y, making it possible to represent integers in the range from 0 to 31; hence the sum $S = X + Y = (25)_{10}$ can also be denoted as a five-bit integer. Note the labeling of individual bits, such that $X = x_4x_3x_2x_1x_0$ and $Y = y_4y_3y_2y_1y_0$. The figure shows, in a red color, the carries generated during the addition process. For example, a carry of 0 is generated when x_0 and y_0 are added, a carry of 1 is produced when x_1 and y_1 are added, and so on.

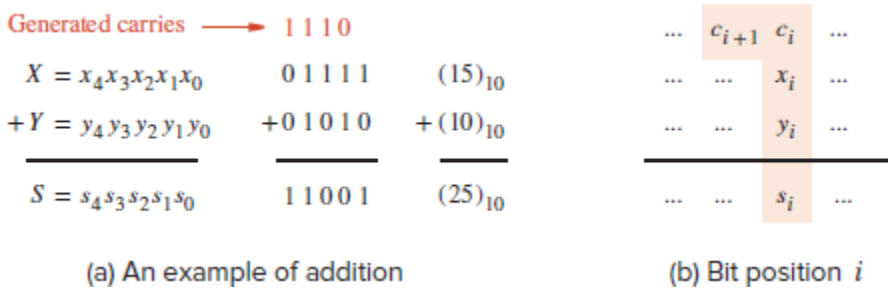

Generated carries → 1 1 1 0
X = x4 x3 x2 x1 x0 0 1 1 1 1 (15)10
+Y = y4 y3 y2 y1 y0 +0 1 0 1 0 +(10)10
S = s4 s3 s2 s1 s0 1 1 0 0 1 (25)10
(a) An example of addition
... ci+1 ci ...
... ... xi ...
... ... yi ...
... ... si ...
(b) Bit position i

Figure 3.2 Addition of multibit numbers.

In Chapter 2 we designed logic circuits by first specifying their behavior in the form of a truth table. This approach is impractical in designing an adder circuit that can add the five-bit numbers in Figure 3.2. The required truth table would have 10 input variables, 5 for each number X and Y. It would have $2^{10} = 1024$ rows! A better approach is to consider the addition of each pair of bits, x_i and y_i, separately.

For bit position 0, there is no carry-in, and hence the addition is the same as for Figure 3.1. For each other bit position i, the addition involves bits x_i and y_i, and a carry-in c_i, as illustrated in Figure 3.2*b*. This observation leads to the design of a logic circuit that has three inputs x_i, y_i, and c_i, and produces the two outputs s_i and c_{i+1}. The required truth table is shown in Figure 3.3*a*. The sum bit, s_i, is the modulo-2 sum of x_i, y_i, and c_i. The *carry-out*, c_{i+1}, is equal to 1 if the sum of x_i, y_i, and c_i is equal to either 2 or 3. Karnaugh maps for these functions are shown in part (*b*) of the figure. For the carry-out function the optimal sum-of-products realization is

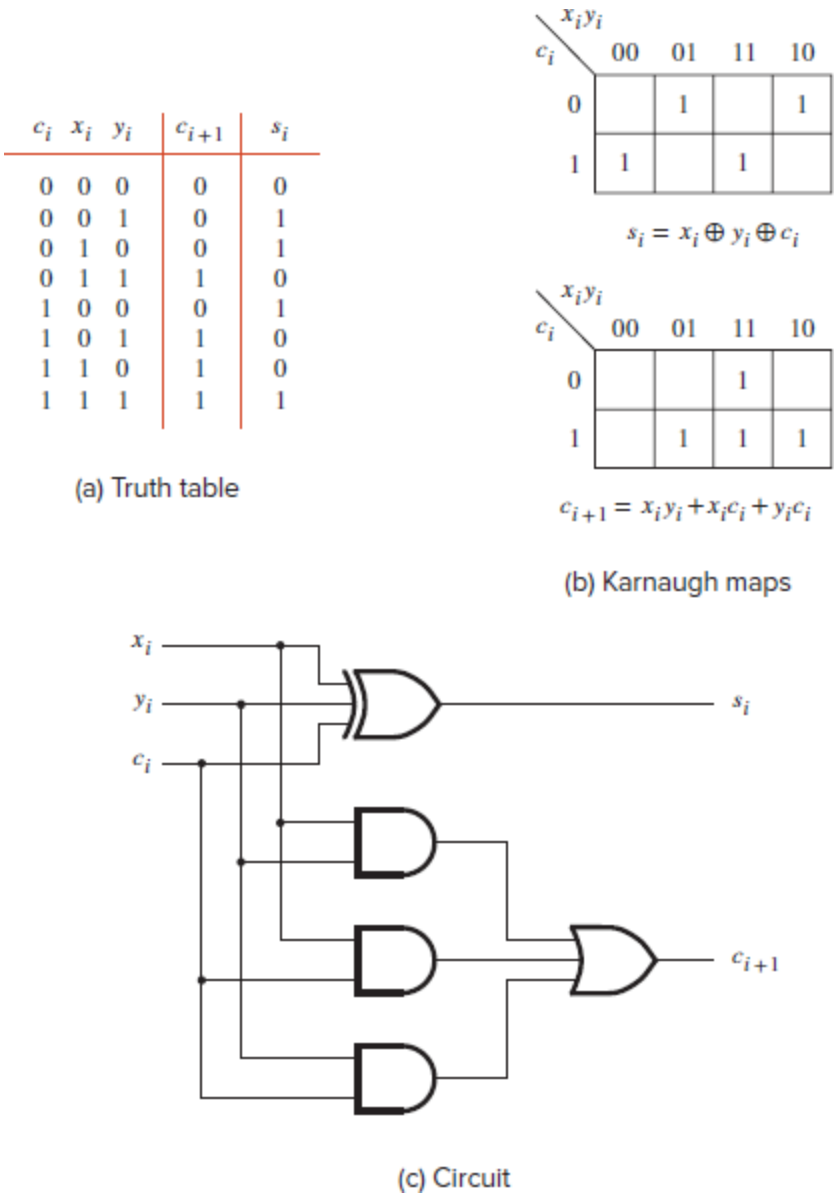


Figure 3.3 Full-adder.

$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

For the s_i function a sum-of-products realization is

$$s_i = \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + \bar{x}_i \bar{y}_i c_i + x_i y_i c_i$$

A more attractive way of implementing this function is by using the XOR gates, as explained below.

Use of XOR Gates

As shown in Chapter 2, the XOR function of two variables is defined as $x_1 \oplus x_2 = \bar{x}_1 x_2 + x_1 \bar{x}_2$. The preceding expression for the sum bit can be manipulated into a form that uses only XOR operations as follows

$$\begin{aligned} s_i &= (\bar{x}_i y_i + x_i \bar{y}_i)\bar{c}_i + (\bar{x}_i \bar{y}_i + x_i y_i)c_i \\ &= (x_i \oplus y_i)\bar{c}_i + \overline{(x_i \oplus y_i)}c_i \\ &= (x_i \oplus y_i) \oplus c_i \end{aligned}$$

The XOR operation is associative; hence we can write

$$s_i = x_i \oplus y_i \oplus c_i$$

Therefore, a three-input XOR operation can be used to realize s_i.

The XOR operation generates as an output a modulo-2 sum of its inputs. Thus, the output is equal to 1 if an odd number of inputs have the value 1, and it is equal to 0 otherwise. For this reason the XOR is sometimes referred to as the *odd* function. Observe that the XOR has no minterms that can be combined into a larger product term, as evident from the checkerboard pattern for function s_i in the map in Figure 3.3*b*. The logic circuit implementing the truth table in Figure 3.3*a* is given in Figure 3.3*c*. This circuit is known as a *full-adder*.

Another interesting feature of XOR gates is that a two-input XOR gate can be thought of as using one input as a control signal that determines whether the true or complemented value of the other input will be passed through the gate as the output value. This is clear from the definition of XOR, where $x_i \oplus y_i = \bar{x}y + x\bar{y}$. Consider x to be the control input. Then if $x = 0$, the output will be equal to the value of y. But if $x = 1$, the output will be equal to the complement of y. In the derivation above, we used algebraic manipulation to derive $s_i = (x_i \oplus y_i) \oplus c_i$. We could have obtained the same expression immediately by making the following observation. In the top half of the truth table in Figure 3.3*a*, c_i is equal to 0, and the sum function s_i is the XOR of x_i and y_i. In the bottom half of the table, c_i is equal to 1, while s_i is the complemented version of its top half. This observation leads directly to our expression using 2 two-input XOR operations. We will

encounter an important example of using XOR gates to pass true or complemented signals under the control of another signal in Section 3.3.3.

In the preceding discussion we encountered the complement of the XOR operation, which we denoted as $\overline{x \oplus y}$. This operation is used so commonly that it is given the distinct name *XNOR*. A special symbol, $\odot$, is often used to denote the XNOR operation, namely

$$x \odot y = \overline{x \oplus y}$$

The XNOR is sometimes also referred to as the *coincidence* operation because it produces the output of 1 when its inputs coincide in value; that is, they are both 0 or both 1.

3.2.1 DECOMPOSED FULL-ADDER

In view of the names used for the circuits, one can expect that a full-adder can be constructed using half-adders. This can be accomplished by creating a multilevel circuit given in Figure 3.4. It uses two half-adders to form a full-adder. The reader should verify the functional correctness of this circuit.

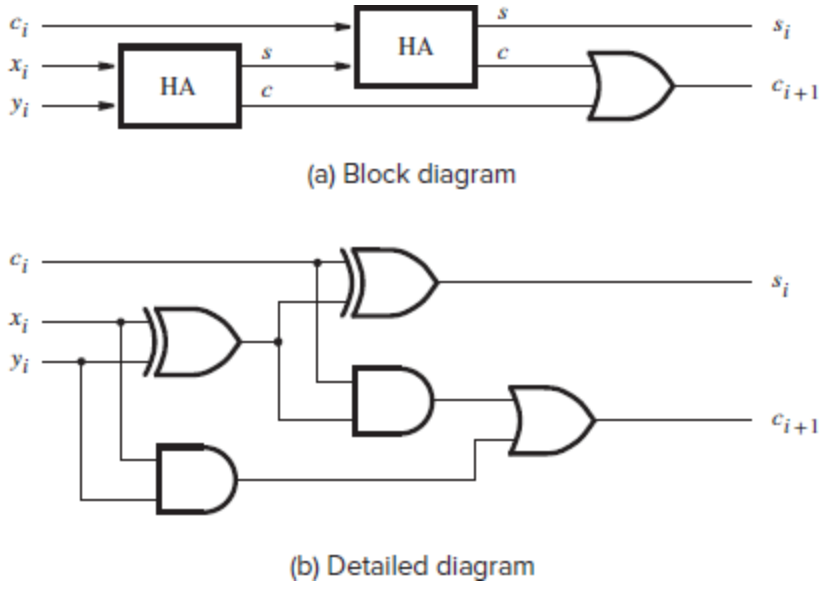


Figure 3.4 A decomposed implementation of the full-adder circuit.

3.2.2 RIPPLE-CARRY ADDER

To perform addition by hand, we start from the least-significant digit and add pairs of digits, progressing to the most-significant digit. If a carry is produced in position *i*, then this carry is added to the operands in position *i*+1. The same arrangement can be used in a logic circuit that performs addition. For each bit position we can use a full-adder circuit, connected as shown in Figure 3.5. Note that to be consistent with the customary way of writing numbers, the least-significant bit position is on the right. Carries that are produced by the full-adders propagate to the left.

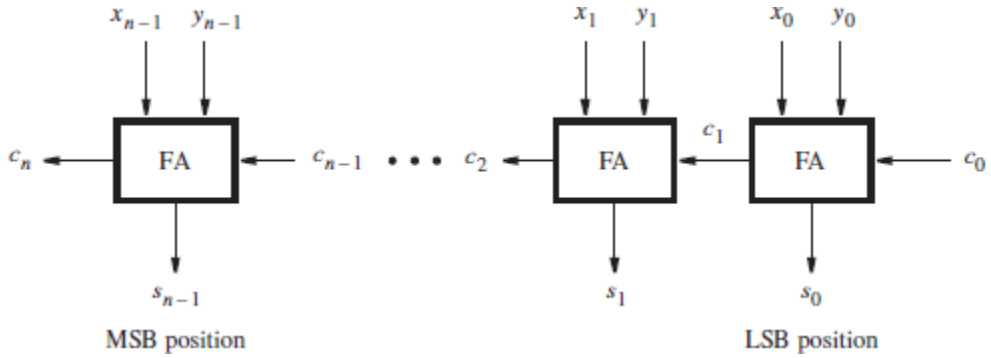


Figure 3.5 An *n*-bit ripple-carry adder.

When the operands *X* and *Y* are applied as inputs to the adder, it takes some time before the output sum, *S*, is valid. Each full-adder introduces a certain delay before its s_i and c_{i+1} outputs are valid. Let this delay be denoted as Δt. Thus the carry-out from the first stage, c_1, arrives at the second stage Δt after the application of the x_0 and y_0 inputs. The carry-out from the second stage, c_2, arrives at the third stage with a $2\Delta t$ delay, and so on. The signal c_{n-1} is valid after a delay of $(n-1)\Delta t$, which means that the complete sum is available after a delay of $n\Delta t$. Because of the way the carry signals "ripple" through the full-adder stages, the circuit in Figure 3.5 is called a *ripple-carry adder*.

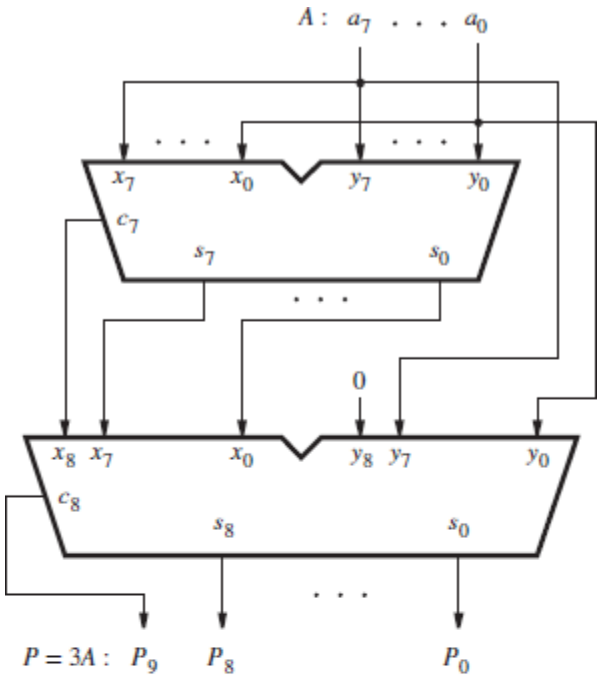
The delay incurred to produce the final sum and carry-out in a ripple-carry adder depends on the size of the numbers. When 32- or 64-bit numbers are used, this delay may become unacceptably high. Because the circuit in each full-adder leaves little room for a drastic reduction in the delay, it may be necessary to seek different structures for implementation of *n*-bit adders. We will discuss a technique for building high-speed adders in Section 3.4.

So far we have dealt with unsigned integers only. The addition of such numbers does not require a carry-in for stage 0. In Figure 3.5 we included c_0 in the diagram so that the ripple-carry adder can also be used for subtraction of numbers, as we will see in Section 3.3.

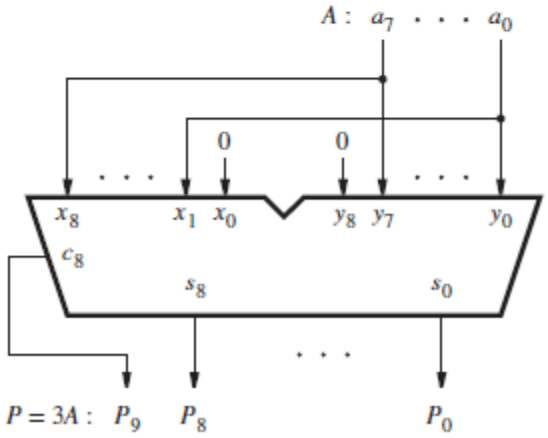
3.2.3 DESIGN EXAMPLE

Suppose that we need a circuit that multiplies an eight-bit unsigned number by 3. Let $A = a_7a_6 \cdots a_1a_0$ denote the number and $P = p_9p_8 \cdots p_1p_0$ denote the product $P = 3A$. Note that 10 bits are needed to represent the product.

A simple approach to design the required circuit is to use two ripple-carry adders to add three copies of the number A, as illustrated in Figure 3.6*a*. The symbol that denotes each adder is a commonly used graphical symbol for adders. The letters x_i, y_i, s_i, and c_i indicate the meaning of the inputs and outputs according to Figure 3.5. The first adder produces $A + A = 2A$. Its result is represented as eight sum bits and the carry from the most-significant bit. The second adder produces $2A + A = 3A$. It has to be a nine-bit adder to be able to handle the nine bits of 2A, which are generated by the first adder. Because the y_i inputs have to be driven only by the eight bits of A, the ninth input y_8 is connected to a constant 0.



(a) Naive approach



(b) Efficient design

Figure 3.6 Circuit that multiplies an eight-bit unsigned number by 3.

This approach is straightforward, but not very efficient. Because $3A = 2A + A$, we can observe that $2A$ can be generated by shifting the bits of A one bit-position to the left, which gives the bit pattern $a_7a_6a_5a_4a_3a_2a_1a_0$0. According to Equation 3.1, this pattern is equal to $2A$. Then a single ripple-carry adder suffices for implementing $3A$, as shown in Figure 3.6*b*. This is essentially the same circuit as the second adder in part (*a*) of the figure.

Note that the input x_0 is connected to a constant 0. Note also that in the second adder in part (*a*) of the figure the value of x_0 is always 0, even though it is driven by the least-significant bit, s_0, of the sum of the first adder. Because $x_0 = y_0 = a_0$ in the first adder, the sum bit s_0 will be 0, whether a_0 is 0 or 1.

3.3 Signed Numbers

In the decimal system the sign of a number is indicated by a + or - symbol to the left of the most-significant digit. In the binary system the *sign* of a number is denoted by the leftmost bit. For a positive number the leftmost bit is equal to 0, and for a negative number it is equal to 1. Therefore, in signed numbers the leftmost bit represents the sign, and the remaining n - 1 bits represent the magnitude, as illustrated in Figure 3.7. It is important to note the difference in the location of the most-significant bit (MSB). In unsigned numbers all bits represent the magnitude of a number; hence all n bits are *significant* in defining the magnitude. Therefore, the MSB is the leftmost bit, b_{n-1}. In signed numbers there are n - 1 significant bits, and the MSB is in bit position b_{n-2}.

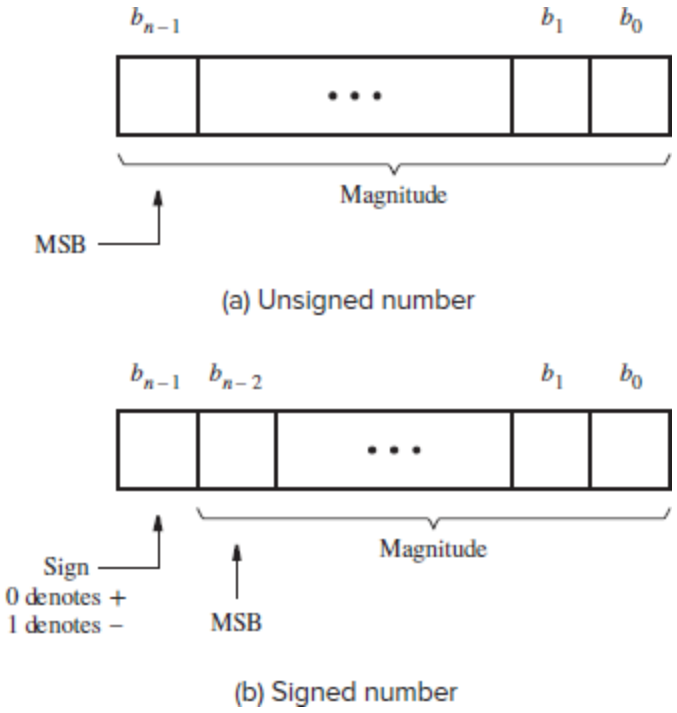


Figure 3.7 Formats for representation of integers.

3.3.1 NEGATIVE NUMBERS

Positive numbers are represented using the positional number representation as explained in the previous section. Negative numbers can be represented in three different ways: sign-and-magnitude, 1's complement, and 2's complement.

Sign-and-Magnitude Representation

In the familiar decimal representation, the magnitude of both positive and negative numbers is expressed in the same way. The sign symbol distinguishes a number as being positive or negative. This scheme is called the *sign-and-magnitude* number representation. The same scheme can be used with binary numbers in which case the sign bit is 0 or 1 for positive or negative numbers, respectively. For example, if we use four-bit numbers, then +5 = 0101 and -5 = 1101. Because of its similarity to decimal sign-and-magnitude numbers, this representation is easy to understand. However, as we will see shortly, this representation is not well suited for use in computers. More suitable representations are based on complementary systems, explained below.

1's Complement Representation

In a complementary number system, the negative numbers are defined according to a subtraction operation involving positive numbers. We will consider two schemes for binary numbers: the 1's complement and the 2's complement. In the *1's complement* scheme, an n-bit negative number, K, is obtained by subtracting its equivalent positive number, P, from 2^n - 1; that is, $K = (2^n - 1) - P$. For example, if $n = 4$, then $K = (2^4 - 1) - P = (15)_{10} - P = (1111)_2 - P$. If we convert +5 to a negative, we get -5 = 1111 - 0101 = 1010. Similarly, +3 = 0011 and -3 = 1111 - 0011 = 1100. Clearly, the 1's complement can be obtained simply by complementing each bit of the number, including the sign bit. While 1's complement numbers are easy to derive, they have some drawbacks when used in arithmetic operations, as we will see in the next section.

2's Complement Representation

In the 2's complement scheme, a negative number, K, is obtained by subtracting its equivalent positive number, P, from 2^n; namely, $K = 2^n - P$. Using our four-bit example, -5 = 10000 - 0101 = 1011, and -3 = 10000 - 0011 = 1101. Finding 2's complements in this manner requires performing a subtraction operation that involves borrows. However, we can observe that if K_1 is the 1's complement of P and K_2 is the 2's complement of P, then

$$K_1 = (2^n - 1) - P$$
$$K_2 = 2^n - P$$

It follows that $K_2 = K_1 + 1$. Thus a simpler way of finding a 2's complement of a number is to add 1 to its 1's complement because finding a 1's complement is trivial. This is how 2's complement numbers are obtained in logic circuits that perform arithmetic operations.

The reader will need to develop an ability to find 2's complement numbers quickly. There is a simple rule that can be used for this purpose.

Rule for Finding 2's Complements

Given a number $B = b_{n-1}b_{n-2} \cdots b_1b_0$, its 2's complement, $K = k_{n-1}k_{n-2} \cdots k_1 k_0$, can be found by examining the bits of B from right to left and taking

the following action: copy all bits of B that are 0 and the first bit that is 1; then simply complement the rest of the bits.

For example, if B = 0110, then we copy $k_0 = b_0 = 0$ and $k_1 = b_1 = 1$, and complement the rest so that $k_2 = \bar{b}_2 = 0$ and $k_3 = \bar{b}_3 = 1$. Hence K = 1010. As another example, if B = 10110100, then K = 01001100. We leave the proof of this rule as an exercise for the reader.

Table 3.2 illustrates the interpretation of all 16 four-bit patterns in the three signed-number representations that we have considered. Note that for both sign-and-magnitude representation and for 1's complement representation there are two patterns that represent the value zero. For 2's complement there is only one such pattern. Also, observe that the range of numbers that can be represented with four bits in 2's complement form is -8 to +7, while in the other two representations it is -7 to +7.

Table 3-2 Interpretation of four-bit signed integers.

$b_3\ b_2\ b_1\ b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

Using 2's-complement representation, an n-bit number $B = b_{n-1}b_{n-2} \cdots b_1b_0$ represents the value

$$V(B) = (-b_{n-1} \times 2^{n-1}) + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \qquad [3.2]$$

Thus the largest negative number, 100 … 00, has the value -2^{n-1}. The largest positive number, 011 … 11, has the value $2^{n-1} - 1$.

3.3.2 ADDITION AND SUBTRACTION

To assess the suitability of different number representations, it is necessary to investigate their use in arithmetic operations—particularly in addition and subtraction. We can illustrate the good and bad aspects of each representation by considering very small numbers. We will use four-bit numbers, consisting of a sign bit and three significant bits. Thus the numbers have to be small enough so that the magnitude of their sum can be expressed in three bits, which means that the sum cannot exceed the value 7.

Addition of positive numbers is the same for all three number representations. It is actually the same as the addition of unsigned numbers discussed in Section 3.2. But there are significant differences when negative numbers are involved. The difficulties that arise become apparent if we consider operands with different combinations of signs.

Sign-and-Magnitude Addition

If both operands have the same sign, then the addition of sign-and-magnitude numbers is simple. The magnitudes are added, and the resulting sum is given the sign of the operands. However, if the operands have opposite signs, the task becomes more complicated. Then it is necessary to subtract the smaller number from the larger one. This means that logic circuits that compare and subtract numbers are also needed. We will see shortly that it is possible to perform subtraction without the need for this circuitry. For this reason, the sign-and-magnitude representation is not used in computers.

1's Complement Addition

An obvious advantage of the 1's complement representation is that a negative number is generated simply by complementing all bits of the corresponding positive number. Figure 3.8 shows what happens when two numbers are added. There are four cases to consider in terms of different combinations of signs. As seen in the top half of the figure, the computation of 5 + 2 = 7 and (-5) + 2 = (-3) is straightforward; a simple addition of the operands gives the correct result. Such is not the case with the other two possibilities. Computing 5 + (-2) = 3 produces the bit vector 10010. Because we are dealing with four-bit numbers, there is a carry-out from the

sign-bit position. Also, the four bits of the result represent the number 2 rather than 3, which is a wrong result. Interestingly, if we take the carry-out from the sign-bit position and add it to the result in the least-significant bit position, the new result is the correct sum of 3. This correction is indicated in red in the figure. A similar situation arises when adding (-5) + (-2) = (-7). After the initial addition the result is wrong because the four bits of the sum are 0111, which represents +7 rather than -7. But again, there is a carry-out from the sign-bit position, which can be used to correct the result by adding it in the LSB position, as shown in Figure 3.8.

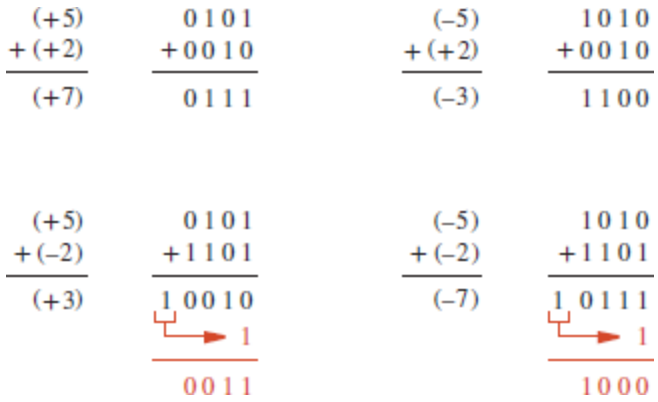


Figure 3.8 Examples of 1's complement addition.

The conclusion from these examples is that the addition of 1's complement numbers may or may not be simple. In some cases a correction is needed, which amounts to an extra addition that must be performed. Consequently, the time needed to add two 1's complement numbers may be twice as long as the time needed to add two unsigned numbers.

2's Complement Addition

Consider the same combinations of numbers as used in the 1's complement example. Figure 3.9 indicates how the addition is performed using 2's complement numbers. Adding 5 + 2 = 7 and (-5) + 2 = (-3) is straightforward. The computation 5 + (-2) = 3 generates the correct four bits of the result, namely 0011. There is a carry-out from the sign-bit position, which we can simply ignore. The fourth case is (-5) + (-2) = (-7). Again, the four bits of the result, 1001, give the correct sum (-7). In this case also, the carry-out from the sign-bit position can be ignored.

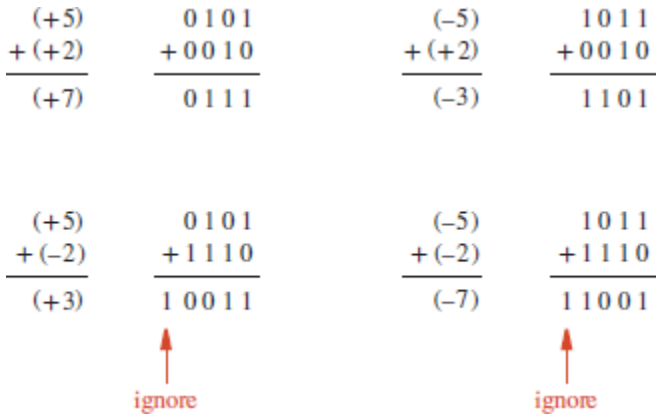


Figure 3.9 Examples of 2's complement addition.

As illustrated by these examples, the addition of 2's complement numbers is very simple. When the numbers are added, the result is always correct. If there is a carry-out from the sign-bit position, it is simply ignored. Therefore, the addition process is the same, regardless of the signs of the operands. It can be performed by an adder circuit, such as the one shown in Figure 3.5. Hence the 2's complement notation is highly suitable for the implementation of addition operations. We will now consider its use in subtraction operations.

2's Complement Subtraction

The easiest way of performing subtraction is to negate the subtrahend and add it to the minuend. This is done by finding the 2's complement of the subtrahend and then performing the addition. Figure 3.10 illustrates the process. The operation 5 - (+2) = 3 involves finding the 2's complement of +2, which is 1110. When this number is added to 0101, the result is 0011 = (+3) and a carry-out from the sign-bit position occurs, which is ignored. A similar situation arises for (-5) - (+2) = (-7). In the remaining two cases there is no carry-out, and the result is correct.

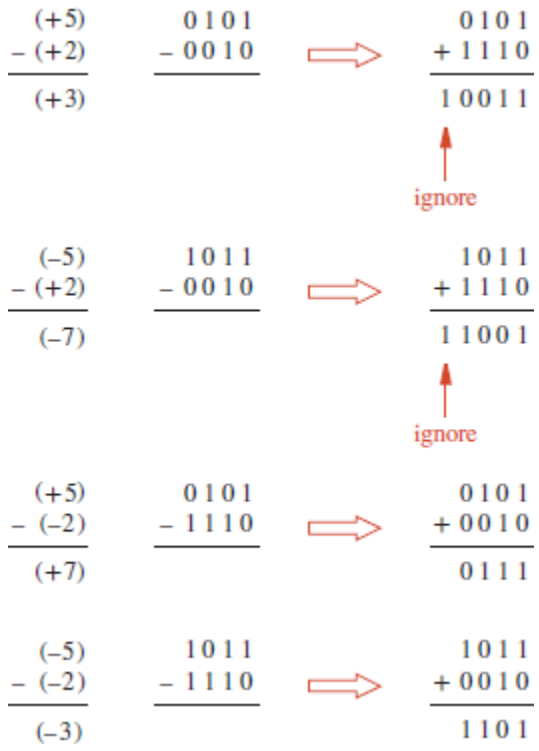


Figure 3.10 Examples of 2's complement subtraction.

As a graphical aid to visualize the addition and subtraction examples in Figures 3.9 and 3.10, we can place all possible four-bit patterns on a modulo-16 circle given in Figure 3.11*a*. If these bit patterns represented unsigned integers, they would be numbers 0 to 15. If they represent 2's-complement integers, then the numbers range from -8 to +7, as shown. The addition operation is done by stepping in the clockwise direction by the magnitude of the number to be added. For example, -5 + 2 is determined by starting at 1011 (= -5) and moving clockwise two steps, giving the result 1101 (= -3). Figure 3.11*b* shows how subtraction can be performed on the modulo-16 circle, using the example 5 - 2 = 3. We can start at 0101 (= +5) and move counterclockwise by two steps, which gives 0011 (= +3). But, we can also use the 2's complement of 2 and add this value by stepping in the clockwise direction as shown in the figure. Since there are 16 numbers on the circle, the value we need to add is 16 - 2 = $(14)_{10}$ = $(1110)_2$.

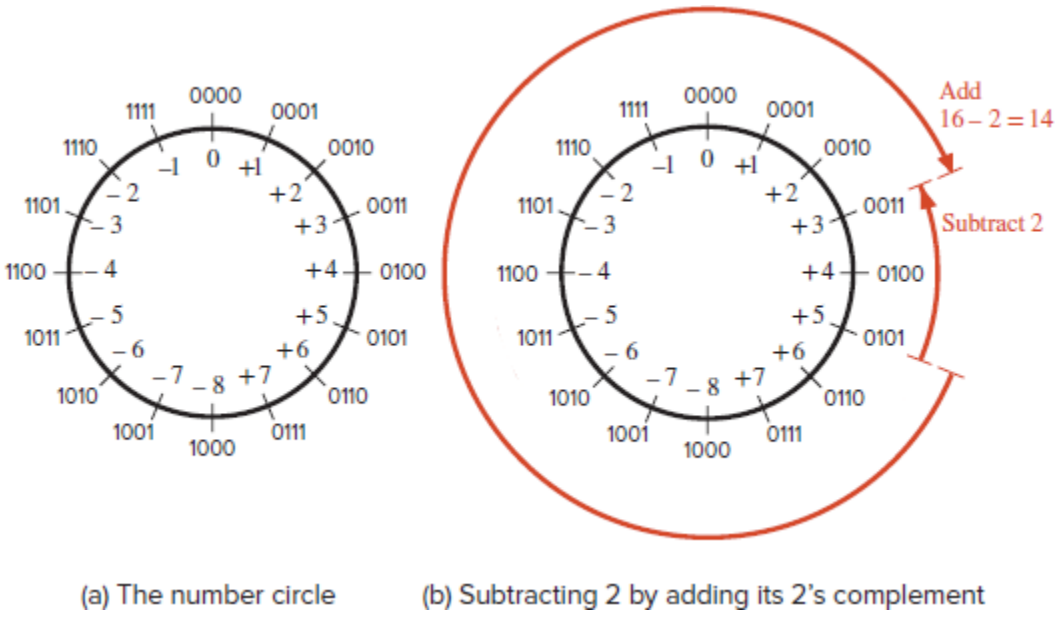


Figure 3.11 Graphical interpretation of four-bit 2's complement numbers.

The key conclusion of this section is that the subtraction operation can be realized as the addition operation, using a 2's complement of the subtrahend, regardless of the signs of the two operands. Therefore, it should be possible to use the same adder circuit to perform both addition and subtraction.

3.3.3 ADDER AND SUBTRACTOR UNIT

The only difference between performing addition and subtraction is that for subtraction it is necessary to use the 2's complement of one operand. Let X and Y be the two operands, such that Y serves as the subtrahend in subtraction. From Section 3.3.1 we know that a 2's complement can be obtained by adding 1 to the 1's complement of Y. Adding 1 in the least-significant bit position can be accomplished simply by setting the carry-in bit c_0 to 1. A 1's complement of a number is obtained by complementing each of its bits. This could be done with NOT gates, but we need a more flexible circuit where we can use the true value of Y for addition and its complement for subtraction.

In Section 3.2 we explained that two-input XOR gates can be used to choose between true and complemented versions of an input value, under the control of the other input. This idea can be applied in the design of the adder/subtractor unit as follows. Assume that there exists a control signal

that chooses whether addition or subtraction is to be performed. Let this signal be called $\overline{\text{Add}}/\text{Sub}$. Also, let its value be 0 for addition and 1 for subtraction. To indicate this fact, we placed a bar over Add. This is a commonly used convention, where a bar over a name means that the action specified by the name is to be taken if the control signal has the value 0. Now let each bit of Y be connected to one input of an XOR gate, with the other input connected to $\overline{\text{Add}}/\text{Sub}$. The outputs of the XOR gates represent Y if $\overline{\text{Add}}/\text{Sub} = 0$, and they represent the 1's complement of Y if $\overline{\text{Add}}/\text{Sub} = 1$. This leads to the circuit in Figure 3.12. The main part of the circuit is an n-bit adder, which can be implemented using the ripple-carry structure of Figure 3.5. Note that the control signal $\overline{\text{Add}}/\text{Sub}$ is also connected to the carry-in c_0. This makes $c_0 = 1$ when subtraction is to be performed, thus adding the 1 that is needed to form the 2's complement of Y. When the addition operation is performed, we will have $c_0 = 0$.

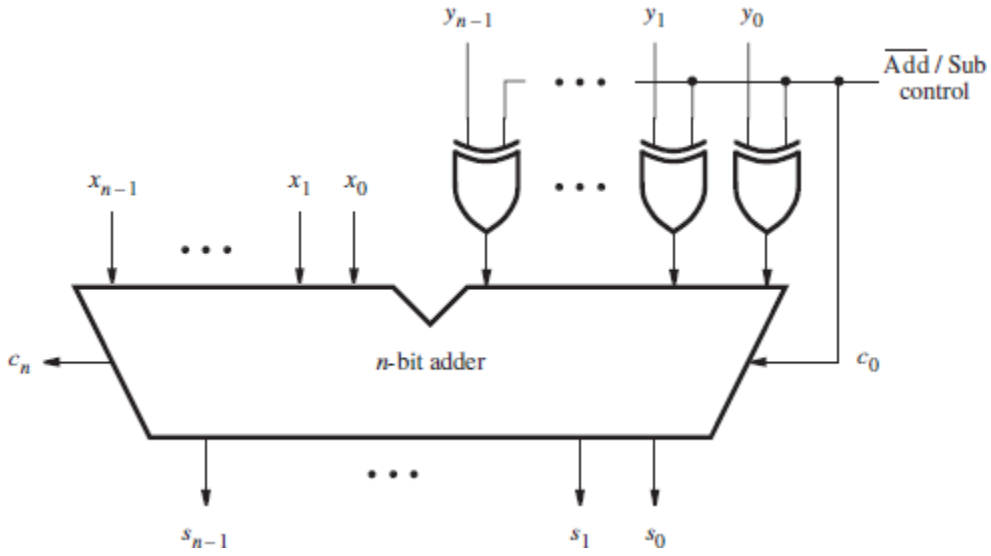


Figure 3.12 Adder/subtractor unit.

The combined adder/subtractor unit is a good example of an important concept in the design of logic circuits. It is useful to design circuits to be as flexible as possible and to exploit common portions of circuits for as many tasks as possible. This approach minimizes the number of gates needed to implement such circuits, and it reduces the wiring complexity substantially.

3.3.4 RADIX-COMPLEMENT SCHEMES*

The 2's complement scheme is just a special case of radix-complement schemes which we discuss in this section. This general discussion can be skipped without loss of continuity in the context of computer technology.

The idea of performing a subtraction operation by addition of a complement of the subtrahend is not restricted to binary numbers. We can gain some insight into the workings of the 2's complement scheme by considering its counterpart in the decimal number system. Consider the subtraction of two-digit decimal numbers. Computing a result such as 74 - 33 = 41 is simple because each digit of the subtrahend is smaller than the corresponding digit of the minuend; therefore, no borrow is needed in the computation. But computing 74 - 36 = 38 is not as simple because a borrow is needed in subtracting the least-significant digit. If a borrow occurs, the computation becomes more complicated.

Suppose that we restructure the required computation as follows

$$\begin{aligned} 74 - 36 &= 74 + 100 - 100 - 36 \\ &= 74 + (100 - 36) - 100 \end{aligned}$$

Now two subtractions are needed. Subtracting 36 from 100 still involves borrows. But noting that 100 = 99 + 1, these borrows can be avoided by writing

$$\begin{aligned} 74 - 36 &= 74 + (99 + 1 - 36) - 100 \\ &= 74 + (99 - 36) + 1 - 100 \end{aligned}$$

The subtraction in parentheses does not require borrows; it is performed by subtracting each digit of the subtrahend from 9. We can see a direct correlation between this expression and the one used for 2's complement, as reflected in the circuit in Figure 3.12. The operation (99 - 36) is analogous to complementing the subtrahend Y to find its 1's complement, which is the same as subtracting each bit from 1. Using decimal numbers, we find the *9's complement* of the subtrahend by subtracting each digit from 9. In Figure 3.12 we add the carry-in of 1 to form the 2's complement of Y. In our decimal example we perform (99 -36) + 1 = 64. Here 64 is the 10's complement of 36. For an n-digit decimal number, N, its *10's complement*, K_{10}, is defined as $K_{10} = 10^n - N$, while its 9's complement, K_9, is $K_9 = (10^n - 1) - N$.

Thus the required subtraction (74 - 36) can be performed by addition of the 10's complement of the subtrahend, as in

$$\begin{aligned} 74 - 36 &= 74 + 64 - 100 \\ &= 138 - 100 \\ &= 38 \end{aligned}$$

The subtraction 138 - 100 is trivial because it means that the leading digit in 138 is simply deleted. This is analogous to ignoring the carry-out from the circuit in Figure 3.12, as discussed for the subtraction examples in Figure 3.10.

Example 3.1

suppose that A and B are n-digit decimal numbers. Using the above 10's-complement approach, B can be subtracted from A as follows:

$$A - B = A + (10^n - B) - 10^n$$

If $A \geq B$, then the operation $A + (10^n - B)$ produces a carry-out of 1. This carry is equivalent to 10^n; hence it can be simply ignored.

But if $A < B$, then the operation $A + (10^n - B)$ produces a carry-out of 0. Let the result obtained be M, so that

$$A - B = M - 10^n$$

We can rewrite this as

$$10^n - (B - A) = M$$

The left side of this equation is the 10's complement of $(B - A)$. The 10's complement of a positive number represents a negative number that has the same magnitude. Hence M correctly represents the negative value obtained from the computation $A - B$ when $A < B$. This concept is illustrated in the examples that follow.

Example 3.2

when dealing with binary signed numbers we use 0 in the leftmost bit position to denote a positive number and 1 to denote a negative number. If we wanted to build hardware that operates on signed decimal numbers, we could use a similar approach. Let 0 in the leftmost digit position denote a positive number and let 9 denote a negative number. Note that 9 is the 9's complement of 0 in the decimal system, just as 1 is the 1's complement of 0 in the binary system.

Thus, using three-digit signed numbers, $A = 045$ and $B = 027$ are positive numbers with magnitudes 45 and 27, respectively. The number B can be subtracted from A as follows

$$\begin{aligned} A - B &= 045 - 027 \\ &= 045 + 1000 - 1000 - 027 \\ &= 045 + (999 - 027) + 1 - 1000 \\ &= 045 + 972 + 1 - 1000 \\ &= 1018 - 1000 \\ &= 018 \end{aligned}$$

This gives the correct answer of +18.

Next consider the case where the minuend has lower value than the subtrahend. This is illustrated by the computation

$$\begin{aligned} B - A &= 027 - 045 \\ &= 027 + 1000 - 1000 - 045 \\ &= 027 + (999 - 045) + 1 - 1000 \\ &= 027 + 954 + 1 - 1000 \\ &= 982 - 1000 \end{aligned}$$

From this expression it appears that we still need to perform the subtraction 982 - 1000. But as seen in Example 3.1, this can be rewritten as

$$\begin{aligned} 982 &= 1000 + B - A \\ &= 1000 - (A - B) \end{aligned}$$

Therefore, 982 is the negative number that results when forming the 10's complement of (A - B). From the previous computation we know that (A - B) = 018, which denotes +18. Thus the signed number 982 is the 10's complement representation of -18, which is the required result.

These examples illustrate that signed numbers can be subtracted without using a subtraction operation that involves borrows. The only subtraction needed is in forming the 9's complement of the subtrahend, in which case each digit is simply subtracted from 9. Thus a circuit that forms the 9's complement, combined with a normal adder circuit, will suffice for both addition and subtraction of decimal signed numbers. A key point is that the hardware needs to deal only with n digits if n-digit numbers are used. Any carry that may be generated from the leftmost digit position is simply ignored.

The concept of subtracting a number by adding its radix-complement is general. If the radix is r, then the r's complement, K_r, of an n-digit number, N, is determined as $K_r = r^n - N$. The $(r - 1)$'s complement, K_{r-1}, is defined as $K_{r-1} = (r^n - 1) - N$; it is computed simply by subtracting each digit of N from the value $(r - 1)$. The $(r - 1)$'s complement is referred to as the *diminished-radix complement*. Circuits for forming the $(r - 1)$'s complements are simpler than those for general subtraction that involves borrows. The circuits are particularly simple in the binary case, where the 1's complement requires just inverting each bit.

Example 3.3

In Figure 3.10 we illustrated the subtraction operation on binary numbers given in 2's-complement representation. Consider the computation (+5) - (+2) = (+3), using the approach discussed above. Each number is represented by a four-bit pattern. The value 2^4 is represented as 10000. Then

$$\begin{aligned} 0101 - 0010 &= 0101 + (10000 - 0010) - 10000 \\ &= 0101 + (1111 - 0010) + 1 - 10000 \\ &= 0101 + 1101 + 1 - 10000 \\ &= 10011 - 10000 \\ &= 0011 \end{aligned}$$

Because 5 > 2, there is a carry from the fourth bit position. It represents the value 2^4, denoted by the pattern 10000.

Example 3.4

Consider now the computation (+2) - (+5) = (-3), which gives

$$\begin{aligned} 0010 - 0101 &= 0010 + (10000 - 0101) - 10000 \\ &= 0010 + (1111 - 0101) + 1 - 10000 \\ &= 0010 + 1010 + 1 - 10000 \\ &= 1101 - 10000 \end{aligned}$$

Because 2 < 5, there is no carry from the fourth bit position. The answer, 1101, is the 2's-complement representation of -3. Note that

$$\begin{aligned} 1101 &= 10000 + 0010 - 0101 \\ &= 10000 - (0101 - 0010) \\ &= 10000 - 0011 \end{aligned}$$

indicating that 1101 is the 2's complement of 0011 (+3).

Example 3.5

Finally, consider the case where the subtrahend is a negative number. The computation (+5) - (-2) = (+7) is done as follows

$$\begin{aligned} 0101 - 1110 &= 0101 + (10000 - 1110) - 10000 \\ &= 0101 + (1111 - 1110) + 1 - 10000 \\ &= 0101 + 0001 + 1 - 10000 \\ &= 0111 - 10000 \end{aligned}$$

While 5 > (-2), the pattern 1110 is greater than the pattern 0101 when the patterns are treated as unsigned numbers. Therefore, there is no carry from the fourth bit position. The answer 0111 is the 2's complement representation of +7. Note that

$$\begin{aligned} 0111 &= 10000 + 0101 - 1110 \\ &= 10000 - (1110 - 0101) \\ &= 10000 - 1001 \end{aligned}$$

and 1001 represents -7.

3.3.5 ARITHMETIC OVERFLOW

The result of addition or subtraction is supposed to fit within the significant bits used to represent the numbers. If n bits are used to represent signed numbers, then the result must be in the range -2^{n-1} to $2^{n-1} - 1$. If the result does not fit in this range, then we say that *arithmetic overflow* has occurred. To ensure the correct operation of an arithmetic circuit, it is important to be able to detect the occurrence of overflow.

Figure 3.13 presents the four cases where 2's-complement numbers with magnitudes of 7 and 2 are added. Because we are using four-bit numbers, there are three significant bits, $b_{2\text{-}0}$. When the numbers have opposite signs, there is no overflow. But if both numbers have the same sign, the magnitude of the result is 9, which cannot be represented with just three significant bits; therefore, overflow occurs. The key to determining whether overflow occurs is the carry-out from the MSB position, called c_3 in the figure, and from the sign-bit position, called c_4. The figure indicates that overflow occurs when these carry-outs have different values, and a correct sum is produced when they have the same value. Indeed, this is true in general for both addition and subtraction of 2's-complement numbers. As a quick check of this statement, consider the examples in Figure 3.9 where the numbers are small enough so that overflow does not occur in any case. In the top two examples in the figure, there is a carry-out of 0 from both sign and MSB positions. In the bottom two examples, there is a carry-out of 1 from both positions. Therefore, for the examples in Figures 3.9 and 3.13, the occurrence of overflow is detected by

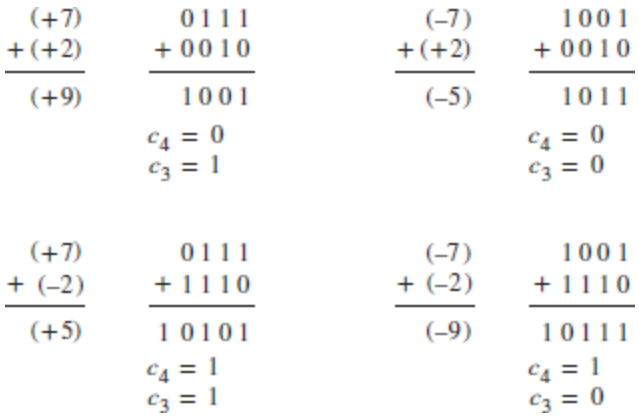


Figure 3.13 Examples for determination of overflow.

$$\begin{aligned}\text{Overflow} &= c_3\bar{c}_4 + \bar{c}_3 c_4 \\ &= c_3 \oplus c_4\end{aligned}$$

For n-bit numbers we have

$$\text{Overflow} = c_{n-1} \oplus c_n$$

Thus the circuit in Figure 3.12 can be modified to include overflow checking with the addition of one XOR gate.

An alternative and more intuitive way of detecting the arithmetic overflow is to observe that overflow occurs if both summands have the same sign but the resulting sum has a different sign. Let $X = x_3x_2x_1x_0$ and $Y = y_3y_2y_1y_0$ represent four-bit 2's-complement numbers, and let $S = s_3s_2s_1s_0$ be the sum $S = X + Y$. Then

$$\text{Overflow} = x_3y_3\bar{s}_3 + \bar{x}_3\bar{y}_3s_3$$

The carry-out and overflow signals indicate whether the result of a given addition is too large to fit into the number of bits available to represent the sum. The carry-out is meaningful only when unsigned numbers are involved, while the overflow is meaningful only in the case of signed numbers. In a typical computer, it is prudent to use the same adder circuits for dealing with both unsigned and signed operands, thus reducing the amount of circuitry required. This means that both the carry-out and overflow signals should be generated, as we have discussed. Then, a program instruction that specifies unsigned operands can use the carry-out signal, while an instruction that has signed operands can use the overflow signal.

3.3.6 PERFORMANCE ISSUES

When buying a digital system, such as a computer, the buyer pays particular attention to the performance that the system is expected to provide and to the cost of acquiring the system. Superior performance usually comes at a higher cost. However, a large increase in performance can often be achieved at a modest increase in cost. A commonly used indicator of the value of a system is its *price/performance ratio*.

The addition and subtraction of numbers are fundamental operations that are performed frequently in the course of a computation. The speed with which these operations are performed has a strong impact on the overall performance of a computer. In light of this, let us take a closer look at the speed of the adder/subtractor unit in Figure 3.12. We are interested in the largest delay from the time the operands X and Y are presented as inputs, until the time all bits of the sum S and the final carry-out, c_n, are valid. Most of this delay is caused by the n-bit adder circuit. Assume that the adder is implemented using the ripple-carry structure in Figure 3.5 and that each full-adder stage is the circuit in Figure 3.3*c*. The delay for the carry-out signal in this circuit, Δt, is equal to two gate delays. From Section 3.2.2 we know that the final result of the addition will be valid after a delay of $n\Delta t$, which is equal to $2n$ gate delays. In addition to the delay in the ripple-carry path, there is also a delay in the XOR gates that feed either the true or complemented value of Y to the adder inputs. If this delay is equal to one gate delay, then the total delay of the circuit in Figure 3.12 is $2n + 1$ gate delays. For a large n, say $n = 32$ or $n = 64$, the delay would lead to unacceptably poor performance. Therefore, it is important to find faster circuits to perform addition.

The speed of any circuit is limited by the longest delay along the paths through the circuit. In the case of the circuit in Figure 3.12, the longest delay is along the path from the y_i input, through the XOR gate and through the carry circuit of each adder stage. The longest delay is often referred to as the *critical-path delay*, and the path that causes this delay is called the *critical path*.

3.4 Fast Adders

The performance of a large digital system is dependent on the speed of circuits that form its various functional units. Obviously, better performance can be achieved using faster circuits. This can be accomplished by using superior (usually newer) technology in which the delays in basic gates are reduced. But it can also be accomplished by changing the overall structure of a functional unit, which may lead to even more impressive improvement.

In this section we will discuss an alternative for implementation of an *n*-bit adder, which substantially reduces the time needed to add numbers.

3.4.1 CARRY-LOOKAHEAD ADDER

To reduce the delay caused by the effect of carry propagation through the ripple-carry adder, we can attempt to evaluate quickly for each stage whether the carry-in from the previous stage will have a value 0 or 1. If a correct evaluation can be made in a relatively short time, then the performance of the complete adder will be improved.

From Figure 3.3*b* the carry-out function for stage *i* can be realized as

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

If we factor this expression as

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

then it can be written as

$$c_{i+1} = g_i + p_i c_i \qquad [3.3]$$

where

$$g_i = x_i y_i$$
$$p_i = x_i + y_i$$

The function g_i is equal to 1 when both inputs x_i and y_i are equal to 1, regardless of the value of the incoming carry to this stage, c_i. Since in this case stage *i* is guaranteed to generate a carry-out, *g* is called the *generate* function. The function p_i is equal to 1 when at least one of the inputs x_i and y_i is equal to 1. In this case a carry-out is produced if c_i = 1. The effect is that the carry-in of 1 is propagated through stage *i*; hence p_i is called the *propagate* function.

Expanding the expression 3.3 in terms of stage *i* - 1 gives

$$\begin{aligned} c_{i+1} &= g_i + p_i(g_{i-1} + p_{i-1} c_{i-1}) \\ &= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \end{aligned}$$

The same expansion for other stages, ending with stage 0, gives

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0 \qquad [3.4]$$

This expression represents a two-level AND-OR circuit in which c_{i+1} is evaluated very quickly. An adder based on this expression is called a *carry-lookahead adder*.

To appreciate the physical meaning of expression 3.4, it is instructive to consider its effect on the construction of a fast adder in comparison with the details of the ripple-carry adder. We will do so by examining the detailed structure of the two stages that add the least-significant bits, namely, stages 0 and 1. Figure 3.14 shows the first two stages of a ripple-carry adder in which the carry-out functions are implemented as indicated in expression 3.3. Each stage is essentially the circuit from Figure 3.3*c* except that an extra OR gate is used (which produces the p_i signal), instead of an AND gate because we factored the sum-of-products expression for c_{i+1}.

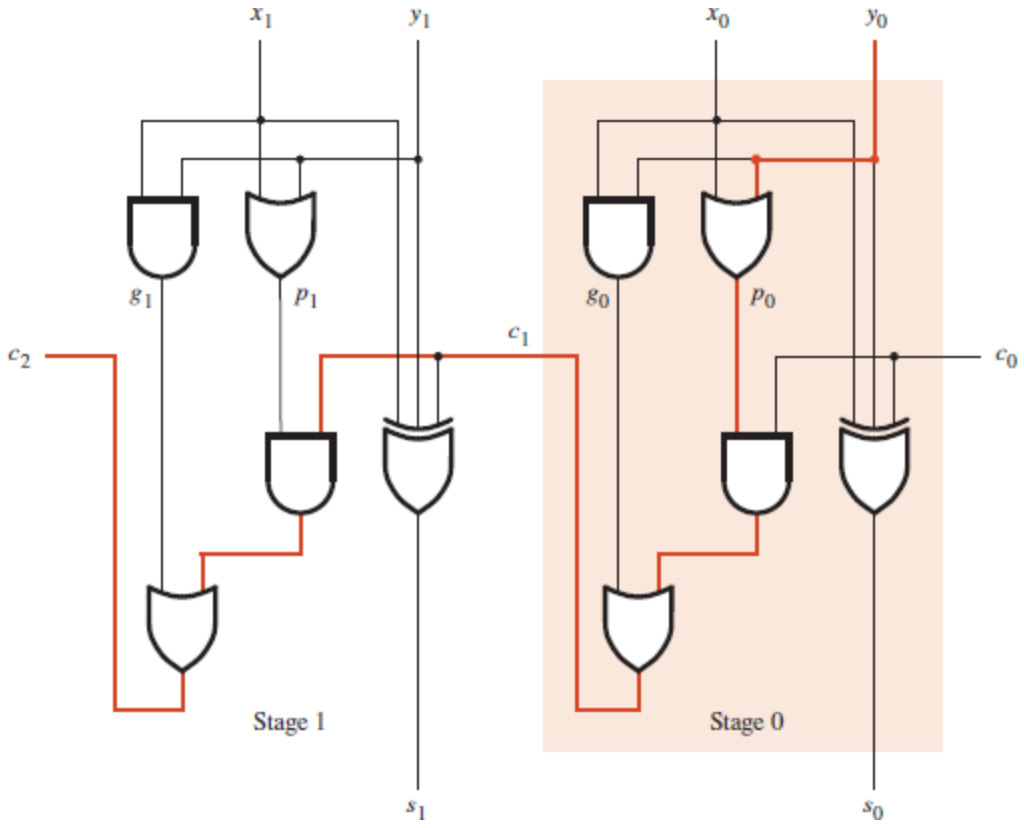


Figure 3.14 A ripple-carry adder based on expression 3.3.

The slow speed of the ripple-carry adder is caused by the long path along which a carry signal must propagate. In Figure 3.14 the critical path is from inputs x_0 and y_0 to the output c_2. It passes through five gates, as highlighted in red. The path in other stages of an n-bit adder is the same as in stage 1. Therefore, the total number of gate delays along the critical path is $2n + 1$.

Figure 3.15 gives the first two stages of the carry-lookahead adder, using expression 3.4 to implement the carry-out functions. Thus

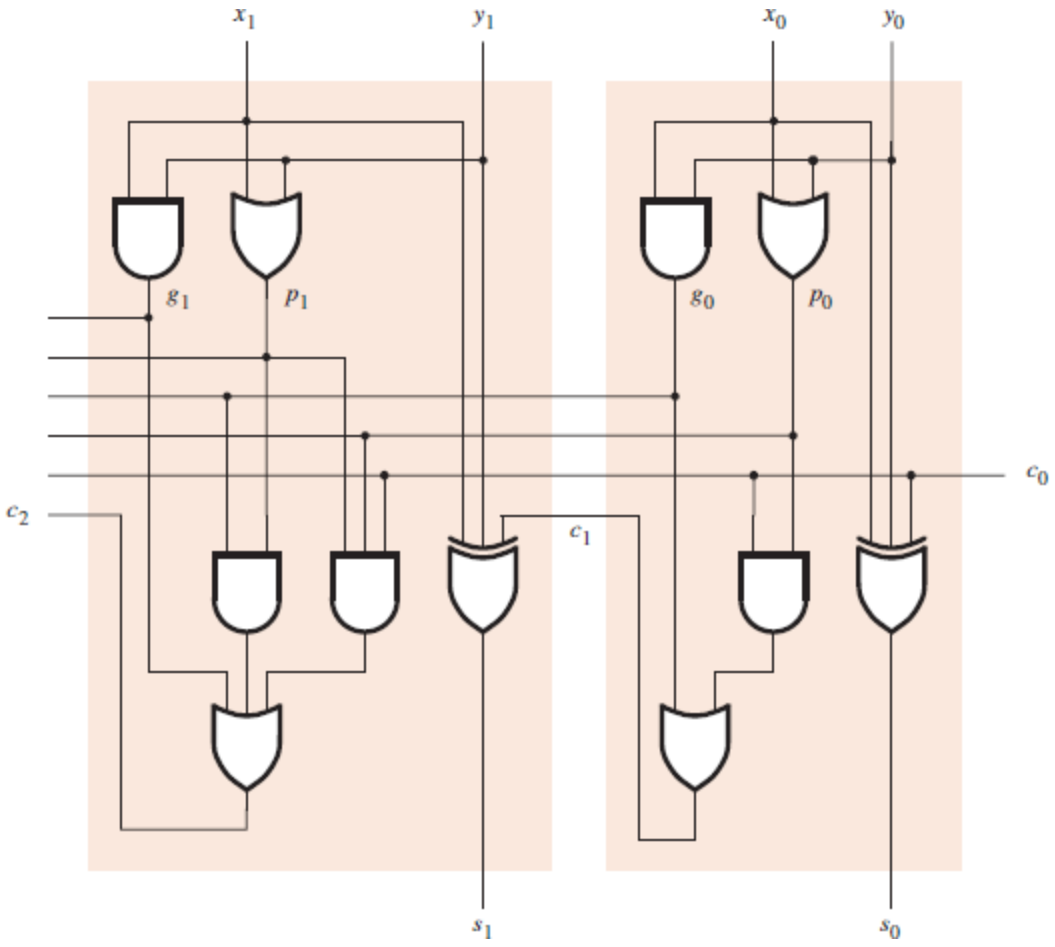


Figure 3.15 The first two stages of a carry-lookahead adder.

$$c_1 = g_0 + p_0 c_0$$
$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

This circuit does not have the long ripple-carry path that is present in Figure 3.14. Instead, all carry signals are produced after three gate delays: one gate delay is needed to produce the generate and propagate signals g_0, g_1, p_0, and

p_1, and two more gate delays are needed to produce c_1 and c_2 concurrently. Extending the circuit to n bits, the final carry-out signal c_n would also be produced after only three gate delays because expression 3.4 is just a large two-level (AND-OR) circuit.

The total delay in the n-bit carry-lookahead adder is four gate delays. The values of all g_i and p_i signals are determined after one gate delay. It takes two more gate delays to evaluate all carry signals. Finally, it takes one more gate delay (XOR) to generate all sum bits. The key to the good performance of the adder is quick evaluation of carry signals.

The complexity of an n-bit carry-lookahead adder increases rapidly as n becomes larger. To reduce the complexity, we can use a *hierarchical* approach in designing large adders. Suppose that we want to design a 32-bit adder. We can divide this adder into 4 eight-bit blocks, such that block 0 adds bits 7 … 0, block 1 adds bits 15 … 8, block 2 adds bits 23 … 16, and block 3 adds bits 31 … 24. Then we can implement each block as an eight-bit carry-lookahead adder. The carry-out signals from the four blocks are c_8, c_{16}, c_{24}, and c_{32}. Now we have two possibilities. We can connect the four blocks as four stages in a ripple-carry adder. Thus while carry-lookahead is used within each block, the carries ripple between the blocks. This circuit is illustrated in Figure 3.16.

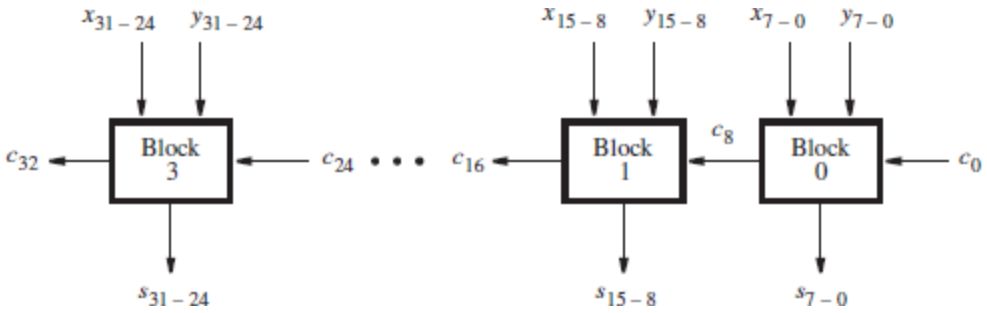


Figure 3.16 A hierarchical carry-lookahead adder with ripple-carry between blocks.

Instead of using a ripple-carry approach between blocks, a faster circuit can be designed in which a second-level carry-lookahead is performed to produce quickly the carry signals between blocks. The structure of this "hierarchical carry-lookahead adder" is shown in Figure 3.17. Each block in the top row includes an eight-bit carry-lookahead adder, based on generate and propagate signals for each stage in the block, as discussed before.

However, instead of producing a carry-out signal from the most-significant bit of the block, each block produces generate and propagate signals for the entire block. Let G_j and P_j denote these signals for each block j. Now G_j and P_j can be used as inputs to a second-level carry-lookahead circuit at the bottom of Figure 3.17, which evaluates all carries between blocks. We can derive the block generate and propagate signals for block 0 by examining the expression for c_8

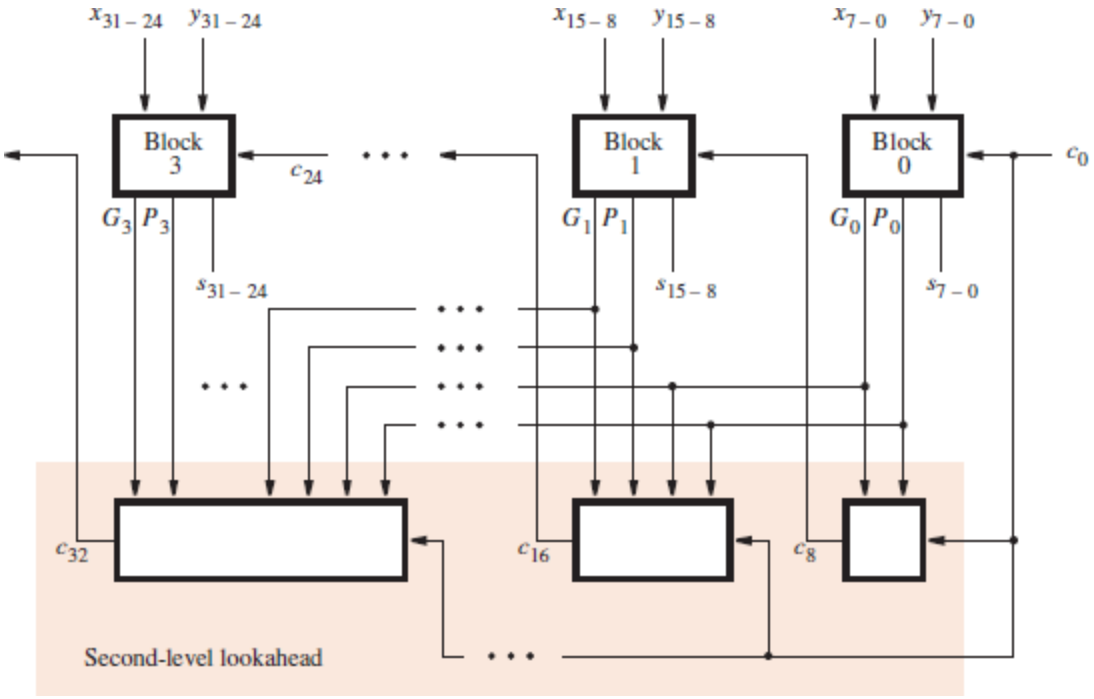


Figure 3.17 A hierarchical carry-lookahead adder.

$$c_8 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0$$

The last term in this expression specifies that, if all eight propagate functions are 1, then the carry-in c_0 is propagated through the entire block. Hence

$$P_0 = p_7p_6p_5p_4p_3p_2p_1p_0$$

The rest of the terms in the expression for c_8 represent all other cases when the block produces a carry-out. Thus

$$G_0 = g_7 + p_7 g_6 + p_7 p_6 g_5 + \cdots + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0$$

The expression for c_8 in the hierarchical adder is given by

$$c_8 = G_0 + P_0 c_0$$

For block 1 the expressions for G_1 and P_1 have the same form as for G_0 and P_0 except that each subscript i is replaced by $i + 8$. The expressions for G_2, P_2, G_3, and P_3 are derived in the same way. The expression for the carry-out of block 1, c_{16}, is

$$\begin{aligned} c_{16} &= G_1 + P_1 c_8 \\ &= G_1 + P_1 G_0 + P_1 P_0 c_0 \end{aligned}$$

Similarly, the expressions for c_{24} and c_{32} are

$$\begin{aligned} c_{24} &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0 \\ c_{32} &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0 \end{aligned}$$

Using this scheme, it takes two more gate delays to produce the carry signals c_8, c_{16}, c_{24}, and c_{32} than the time needed to generate the G_j and P_j functions. Therefore, since G_j and P_j require three gate delays, c_8, c_{16}, c_{24}, and c_{32} are available after five gate delays. The time needed to add two 32-bit numbers involves these five gate delays plus two more to produce the internal carries in blocks 1, 2, and 3, plus one more gate delay (XOR) to generate each sum bit. This gives a total of eight gate delays.

In Section 3.3.6 we determined that it takes $2n+1$ gate delays to add two numbers using a ripple-carry adder. For 32-bit numbers this implies 65 gate delays. It is clear that the carry-lookahead adder offers a large performance improvement. The trade-off is much greater complexity of the required circuit.

Technology Considerations

The preceding delay analysis assumes that gates with any number of inputs can be used. But, the number of gate inputs, referred to as the *fan-in* of the gate, has to be limited in practice as we discuss in Appendix B.

Therefore the reality of fan-in constraints must be taken into account. To illustrate this problem, consider the expressions for the first eight carries:

$$
\begin{aligned}
c_1 &= g_0 + p_0 c_0 \\
c_2 &= g_1 + p_1 g_0 + p_1 p_0 c_0 \\
&\vdots \\
c_8 &= g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2 \\
&\quad + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0
\end{aligned}
$$

Suppose that the maximum fan-in of the gates is four inputs. Then it is impossible to implement all of these expressions with a two-level AND-OR circuit. The biggest problem is c_8, where one of the AND gates requires nine inputs; moreover, the OR gate also requires nine inputs. To meet the fan-in constraint, we can rewrite the expression for c_8 as

$$
\begin{aligned}
c_8 = {} & (g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4) + [(p_7 p_6 p_5 p_4)(g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0)] \\
& + (p_7 p_6 p_5 p_4)(p_3 p_2 p_1 p_0) c_0
\end{aligned}
$$

To implement this expression we need ten AND gates and three OR gates. The propagation delay in generating c_8 consists of one gate delay to develop all g_i and p_i, two gate delays to produce the sum-of-products terms in parentheses, one gate delay to form the product term in square brackets, and one delay for the final ORing of terms. Hence c_8 is valid after five gate delays, rather than the three gate delays that would be needed without the fan-in constraint.

Because fan-in limitations reduce the speed of the carry-lookahead adder, some devices that are characterized by low fan-in include dedicated circuitry for implementation of fast adders. Examples of such devices include FPGAs, which are described in Appendix B.

3.5 Design of Arithmetic Circuits Using CAD Tools

In this section we show how the arithmetic circuits can be designed by using CAD tools.

3.5.1 DESIGN OF ARITHMETIC CIRCUITS USING SCHEMATIC CAPTURE

An obvious way to design an arithmetic circuit via schematic capture is to draw a schematic that contains the necessary logic gates. For example, to create an *n*-bit adder, we could first draw a schematic that represents a full-adder. Then an *n*-bit ripple-carry adder could be created by drawing a higher-level schematic that connects together *n instances* of the full-adder. A hierarchical schematic created in this manner would look like the circuit shown in Figure 3.5. We could also use this methodology to create an adder/subtractor circuit, such as the circuit depicted in Figure 3.12.

The main problem with this approach is that it is cumbersome, especially when the number of bits is large. This issue is even more apparent if we consider creating a schematic for a carry-lookahead adder. As shown in Section 3.4.1, the carry circuitry in each stage of the carry-lookahead adder becomes increasingly more complex. Hence it is necessary to draw a separate schematic for each stage of the adder. A better approach for creating arithmetic circuits via schematic capture is to use predefined subcircuits.

The schematic capture tools provide a library of graphical symbols that represent basic logic gates. These gates are used to create schematics of relatively simple circuits. In addition to basic gates, most schematic capture tools also provide a library of commonly used circuits, such as adders. Each circuit is provided as a module that can be imported into a schematic and used as part of a larger circuit.

Different vendors of CAD tools have their specific approaches to how schematic capture is performed. Instead of dealing with the schematic capture approach, we will concentrate on the more convenient and flexible approach of using VHDL to design our circuits.

3.5.2 DESIGN OF ARITHMETIC CIRCUITS USING VHDL

We said in Section 3.5.1 that an obvious way to create an *n*-bit adder is to draw a hierarchical schematic that contains *n* full-adders. This approach can also be followed with VHDL, by first creating a VHDL entity for a full-adder and then creating a higher-level entity that uses *n instances* of the full-adder. As a first attempt at designing arithmetic circuits by using

VHDL, we will show how to write the hierarchical code for a ripple-carry adder.

The complete code for a full-adder entity is given in Figure 3.18. It has the inputs *Cin*, *x*, and *y* and produces the outputs *s* and *Cout*. The sum, *s*, and carry-out, *Cout*, are described by logic equations.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT ( Cin, x, y  : IN   STD_LOGIC ;
           s, Cout    : OUT  STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
    s <= x XOR y XOR Cin ;
    Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
END LogicFunc ;
```

Figure 3.18 VHDL code for the full-adder.

We can now create a separate VHDL entity for the ripple-carry adder, which instantiates the *fulladd* entity as a subcircuit. One method of doing this is shown in Figure 3.19. The entity comprises the code for a four-bit ripple-carry adder, named *adder4*. One of the four-bit numbers to be added is represented by the four signals x_3, x_2, x_1, x_0, and the other number is represented by y_3, y_2, y_1, y_0. The sum is represented by s_3, s_2, s_1, s_0. The entity incorporates a carry input, *Cin*, into the least-significant bit position and a carry output, *Cout*, from the most-significant bit position.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder4 IS
    PORT ( Cin            : IN    STD_LOGIC ;
           x3, x2, x1, x0 : IN    STD_LOGIC ;
           y3, y2, y1, y0 : IN    STD_LOGIC ;
           s3, s2, s1, s0 : OUT   STD_LOGIC ;
           Cout           : OUT   STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
    COMPONENT fulladd
        PORT ( Cin, x, y : IN    STD_LOGIC ;
               s, Cout   : OUT   STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
    stage3: fulladd PORT MAP (
        Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;
END Structure ;
```

Figure 3.19 VHDL code for a four-bit adder.

As discussed in Section 2.10.3, signals that are needed internally in an entity, but are not either inputs or outputs, can be declared in the architecture. Three such signals are declared in Figure 3.19 called c_1, c_2, and c_3, which are used as carry-out signals from the first three stages of the adder. The COMPONENT declaration statement, introduced in Section 2.10.3, allows the *fulladd* entity to be used as a component (subcircuit) in the architecture body.

The four-bit adder in Figure 3.19 is described using four *instantiation* statements. Each statement begins with an *instance name*, which can be any legal VHDL name, followed by the colon character. The names must be unique. The least-significant stage in the adder is named *stage0* and the most-significant stage is *stage3*. Observe that in the first three instantiation statements, the signals are listed in the same order as in the *fulladd* COMPONENT, namely, the order *Cin*, *x*, *y*, *s*, *Cout*. It is also possible to list the signal names in other orders by specifying explicitly which signal is to be connected to which port on the component. An example of this style is shown for the *stage3* instance. This style of component instantiation is

known as *named association* in the VHDL jargon, whereas the style used for the other three instances is called *positional association*. Note that for the *stage3* instance, the signal name *Cout* is used as both the name of the component port and the name of the output in the *adder4* entity. This does not cause a problem for the VHDL compiler, because the component port name is always the one on the left side of the => designator.

As discussed in Section 2.10.3, the signal names associated with each instance of the *fulladd* component implicitly specify how the full-adders are connected together. For example, the carry-out of the *stage0* instance is connected to the carry-in of the *stage1* instance. The *fulladd* entity may be included in the same VHDL source-code file as the *adder4* entity, or it may comprise a separate file. The circuit that would be synthesized by a VHDL compiler has the same structure as the one shown in Figure 3.5.

Alternative Style of Code

In Figure 3.19 a component declaration statement for the *fulladd* entity is included in the *adder4* architecture. An alternative approach is to place the component declaration statement in a VHDL *package*. In general, a package allows VHDL constructs to be defined in one source-code file and then used in other files. Two examples of constructs that are often placed in a package are data type declarations and component declarations.

We have already seen an example of using a package for a data type. In Chapter 2 we introduced the package named *std_logic_1164*, which defines the STD_LOGIC signal type. Recall that to access this package, VHDL code must include the statements

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
```

These statements appear in Figures 3.18 and 3.19 because the STD_LOGIC type is used in the code. The first statement provides access to the library named *ieee*. As we discussed in Section 2.10.1, the library represents the location, or *directory*, in the computer file system where the *std_logic_1164* package is stored.

The code in Figure 3.20 defines the package named *fulladd_package*. This code can be stored in a separate VHDL source-code file, or it can be included in the same file used to store the code for the *fulladd* entity. The

VHDL syntax requires that the package declaration have its own LIBRARY and USE clauses; hence they are included in the code. Inside the package the *fulladd* entity is declared as a COMPONENT. When this code is compiled, the*fulladd_package* package is created and stored in the working directory where the code is stored.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE fulladd_package IS
    COMPONENT fulladd
        PORT ( Cin, x, y   : IN     STD_LOGIC ;
               s, Cout     : OUT  STD_LOGIC ) ;
    END COMPONENT ;
END fulladd_package ;
```

Figure 3.20 Declaration of a package.

Any VHDL entity can then use the *fulladd* component as a subcircuit by making use of the *fulladd_package* package. The package is accessed using the two statements

```
LIBRARY work;
USE work.fulladd_package.all ;
```

The library named *work* represents the working directory where the VHDL code that defines the package is stored. This statement is actually not necessary, because the VHDL compiler always has access to the working directory.

Figure 3.21 shows how the code in Figure 3.19 can be rewritten to make use of the *fulladd_package*. The code is the same as that in Figure 3.19 with two exceptions: the extra USE clause is added, and the component declaration statement is deleted in the architecture. The circuits synthesized from the two versions of the code are identical.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder4 IS
    PORT ( Cin          : IN   STD_LOGIC ;
           x3, x2, x1, x0  : IN   STD_LOGIC ;
           y3, y2, y1, y0  : IN   STD_LOGIC ;
           s3, s2, s1, s0  : OUT  STD_LOGIC ;
           Cout         : OUT  STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
    stage3: fulladd PORT MAP (
        Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;
END Structure ;
```

Figure 3.21 A different way of specifying a four-bit adder.

In Figures 3.19 and 3.21, each of the four-bit inputs and the four-bit output of the adder is represented using single-bit signals. A more convenient style of code is to use multibit signals to represent the numbers.

3.5.3 REPRESENTATION OF NUMBERS IN VHDL CODE

Just as a number is represented in a logic circuit as signals on multiple wires, a number is represented in VHDL code as a multibit SIGNAL data object. An example of a multibit signal is

```
SIGNAL C : STD_LOGIC_VECTOR (1 TO 3) ;
```

The STD_LOGIC_VECTOR data type represents a linear array of STD_LOGIC data objects. In VHDL jargon the STD_LOGIC_VECTOR is said to be a subtype of STD_LOGIC. There exists a similar subtype, called BIT_VECTOR, corresponding to the BIT type that was used in Section 2.10. The preceding SIGNAL declaration defines *C* as a three-bit STD_LOGIC signal. It can be used in VHDL code as a three-bit quantity simply by using the name *C*, or else each individual bit can be referred to separately using the names *C*(1), *C*(2), and *C*(3). The syntax 1 TO 3 in the

declaration statement specifies that the leftmost bit in *C* is called *C*(1) and the rightmost bit is called *C*(3). A three-bit signal value can be assigned to *C* as follows:

```
C <= "100" ;
```

The three-bit value is denoted using double quotes, instead of the single quotes used for one-bit values, as in \lquote1\rquote or \lquote0\rquote. The assignment statement results in *C*(1) = 1, *C*(2) = 0, and *C*(3) = 0. The numbering of the bits in the signal *C*, with the highest index used for the leftmost bit, is a natural way of representing signals that are simply grouped together for convenience but do not represent a number. For example, this numbering scheme would be an appropriate way of declaring the three carry signals named c_1, c_2, and c_3 in Figure 3.21. However, when a multibit signal is used to represent a binary number, it makes more sense to number the bits in the opposite way, with the highest index used for the leftmost (most-significant) bit. For this purpose VHDL provides a second way to declare a multibit signal

```
SIGNAL X : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
```

This statement defines *X* as a four-bit STD_LOGIC_VECTOR signal. The syntax 3 DOWNTO 0 specifies that the leftmost (most-significant) bit in *X* is called *X*(3) and the rightmost (least-significant) bit is *X*(0). This scheme is a more natural way of numbering the bits if *X* is to be used in VHDL code to represent a binary number because the index of each bit corresponds to its position in the number. The assignment statement

```
X <= "1100" ;
```

results in *X*(3) = 1, *X*(2) = 1, *X*(1) = 0, and *X*(0) = 0.

Figure 3.22 illustrates how the code in Figure 3.21 can be written to use multibit signals. The data inputs are the four-bit signals *X* and *Y*, and the sum output is the four-bit signal *S*. The intermediate carry signals are declared in the architecture as the three-bit signal *C*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder4 IS
    PORT ( Cin   : IN  STD_LOGIC ;
           X, Y  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), Y(3), S(3), Cout ) ;
END Structure ;
```

Figure 3.22 A four-bit adder defined using multibit signals.

Using hierarchical VHDL code to define large arithmetic circuits can be cumbersome. For this reason, arithmetic circuits are usually implemented in VHDL in a different way, using arithmetic assignment statements and multibit signals.

3.5.4 ARITHMETIC ASSIGNMENT STATEMENTS

If the following signals are defined

```
SIGNAL  X, Y, S  : STD_LOGIC_VECTOR (15 DOWNTO 0) ;
```

then the arithmetic assignment statement

```
S <= X + Y ;
```

represents a 16-bit adder.

In addition to the + operator, which is used for addition, VHDL provides other arithmetic operators. They are listed in Table A.1, in Appendix A. Complete VHDL code that includes the preceding statement is given in Figure 3.23. The *std_logic_1164* package does not specify that STD_LOGIC signals can be used with arithmetic operators. The second package included in the code, named *std_logic_signed*, allows the signals to

be used in this way. When the code in the figure is translated by the VHDL compiler, it generates an adder circuit to implement the + operator.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
    PORT ( X, Y  : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
    S <= X + Y ;
END Behavior ;
```

Figure 3.23 VHDL code for a 16-bit adder.

The code in Figure 3.23 does not include carry-in or carry-out signals. Also, it does not provide the arithmetic overflow signal. One way in which these signals can be added is given in Figure 3.24. The 17-bit signal named *Sum* is defined in the architecture. The extra bit, *Sum*(16), is used for the carry-out from bit-position 15 in the adder. The statement used to assign the sum of *X*, *Y*, and the carry-in, *Cin*, to the *Sum* signal has an unusual syntax. The meaning of the terms in parentheses, (\lquote0\rquote & X) and (\lquote0\rquote & Y), is that a 0 is concatenated to the left of the 16-bit signals *X* and *Y* to create a 17-bit signal. In VHDL the & operator is called the *concatenate* operator. The reader should not confuse this meaning with the more traditional meaning of & in other hardware description languages in which it is the logical AND operator. The concatenate operator is needed because the VHDL compiler requires that the *X* and *Y* operands of the adder have the same bit-width as the resulting *Sum*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
    PORT ( Cin              : IN   STD_LOGIC ;
           X, Y             : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           S                : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           Cout, Overflow   : OUT  STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
    SIGNAL Sum : STD_LOGIC_VECTOR(16 DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + ('0' & Y) + Cin ;
    S <= Sum(15 DOWNTO 0) ;
    Cout <= Sum(16) ;
    Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```

Figure 3.24 The 16-bit adder from Figure 3.23 with carry and overflow signals.

Another detail to observe from the figure is the statement

```
S <= Sum(15 DOWNTO 0) ;
```

This statement assigns the lower 16 bits of *Sum* to the output sum *S*. The next statement assigns the carry-out from the addition, *Sum*(16), to the carry-out signal, *Cout*. The expression for arithmetic overflow was defined in Section 3.3.5 as $c_{n-1} \oplus c_n$. In our case, c_n corresponds to *Sum*(16), but there is no direct way of accessing c_{n-1}, which is the carry-out from bit-position 14. The reader should verify that the expression $X(15) \oplus Y(15) \oplus Sum(15)$ corresponds to c_{n-1}.

The code in Figure 3.24 uses the package *std_logic_signed* to allow the STD_LOGIC signals to be used with arithmetic operators. The *std_logic_signed* package actually uses another package, which is named *std_logic_arith*. This package defines two data types, called SIGNED and UNSIGNED, for use in arithmetic circuits that deal with signed or unsigned numbers. These data types are the same as the STD_LOGIC_VECTOR type; each one is an array of STD_LOGIC signals. The code in Figure 3.24 can be written to directly use the *std_logic_arith* package as shown in Figure 3.25. The multibit signals *X*, *Y*, *S*, and *Sum* have the type SIGNED.

The code is otherwise identical to that in Figure 3.24 and results in the same circuit.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY adder16 IS
    PORT ( Cin              : IN   STD_LOGIC ;
           X, Y             : IN   SIGNED(15 DOWNTO 0) ;
           S                : OUT  SIGNED(15 DOWNTO 0) ;
           Cout, Overflow   : OUT  STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
    SIGNAL Sum : SIGNED(16 DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + ('0' & Y) + Cin ;
    S <= Sum(15 DOWNTO 0) ;
    Cout <= Sum(16);
    Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```

Figure 3.25 Use of the arithmetic package.

It is an arbitrary choice whether to use the *std_logic_signed* package and STD_LOGIC_ VECTOR signals, as in Figure 3.24, or the *std_logic_arith* package and SIGNED signals, as in Figure 3.25. For use with unsigned numbers, there are also two options. We can use the *std_logic_unsigned* package with STD_LOGIC_VECTOR signals or the *std_logic_arith* package with UNSIGNED signals. For our example code in Figures 3.24 and 3.25, the same circuit would be generated whether we assume signed or unsigned numbers. But for unsigned numbers we should not produce a separate *Overflow* output, because the carry-out represents the arithmetic overflow for unsigned numbers.

Before leaving our discussion of arithmetic statements in VHDL, we should mention another signal data type that can be used for arithmetic. The following statement defines the signal *X* as an INTEGER

```
SIGNAL X : INTEGER RANGE –32768 TO 32767 ;
```

For an INTEGER data object, the number of bits is not specified explicitly. Instead, the *range* of numbers to be represented is specified. For a 16-bit

signed integer, the range of representable numbers is -32768 to 32767. An example of using the INTEGER data type in code corresponding to Figure 3.23 is shown in Figure 3.26. No LIBRARY or USE clause appears in the code, because the INTEGER type is predefined in standard VHDL. Although the code in the figure is straightforward, it is more difficult to modify this code to include carry signals and the overflow output shown in Figures 3.24 and 3.25. The method that we used, in which the bits from the signal *Sum* are used to define the carry-out and arithmetic overflow signals, cannot be used for INTEGER objects. To prevent overflow the sum output *S* in Figure 3.26 has a range that is double that of the inputs *X* and *Y*.

```
ENTITY adder16 IS
    PORT ( X, Y  : IN   INTEGER RANGE -32768 TO 32767 ;
           S     : OUT  INTEGER RANGE -65536 TO 65535 ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
    S <= X + Y ;
END Behavior ;
```

Figure 3.26 The 16-bit adder from Figure 3.23 using INTEGER signals.

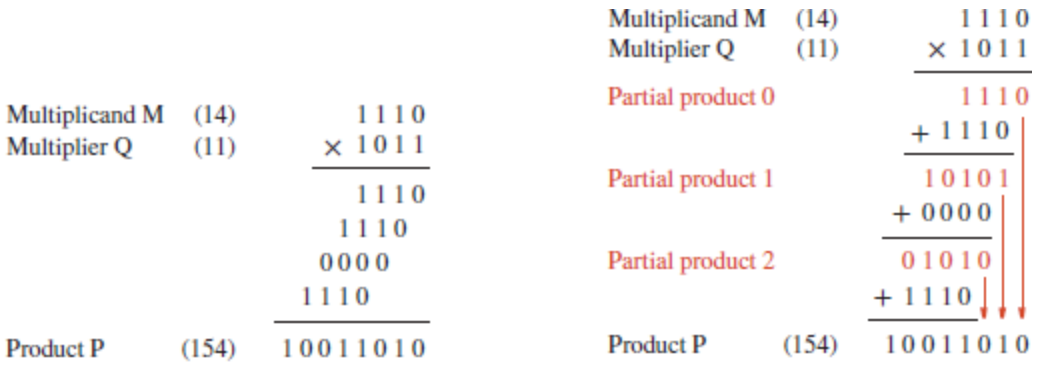
3.6 MULTIPLICATION

Before we discuss the general issue of multiplication, we should note that a binary number, *B*, can be multiplied by 2 simply by adding a zero to the right of its least-significant bit. This effectively moves all bits of *B* to the left, and we say that *B* is *shifted* left by one bit position. Thus if $B = b_{n-1}b_{n-2}\cdots b_1b_0$, then $2 \times B = b_{n-1}b_{n-2}\cdots b_1b_0 0$. (We have already used this fact in Section 3.2.3.) Similarly, a number is multiplied by 2^k by shifting it left by *k* bit positions. This is true for both unsigned and signed numbers.

We should also consider what happens if a binary number is shifted right by *k* bit positions. According to the positional number representation, this action divides the number by 2^k. For unsigned numbers the shifting amounts to adding *k* zeros to the left of the most-significant bit. For example, if *B* is an unsigned number, then $B \div 2 = 0b_{n-1}b_{n-2}\cdots b_2b_1$. Note that bit b_0 is lost when shifting to the right. For signed numbers it is necessary to

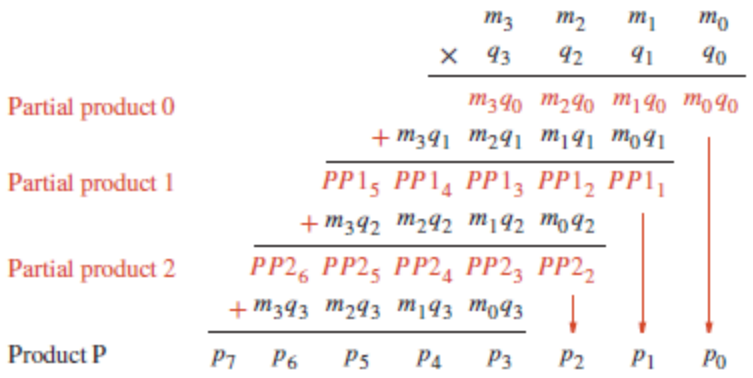
preserve the sign. This is done by shifting the bits to the right and filling from the left with the value of the sign bit. Hence if B is a signed number, then $B \div 2 = b_{n-1}b_{n-1}b_{n-2} \cdots b_2 b_1$. For instance, if B = 011000 = $(24)_{10}$, then $B \div 2 = 001100 = (12)_{10}$ and $B \div 4 = 000110 = (6)_{10}$. Similarly, if B = 101000 = -$(24)_{10}$, then $B \div 2 = 110100 = -(12)_{10}$ and $B \div 4 = 111010 = -(6)_{10}$.

Now we can turn our attention to the general task of multiplication. Two binary numbers can be multiplied using the same method as we use for decimal numbers. We will focus our discussion on multiplication of unsigned numbers. Figure 3.27*a* shows how multiplication is performed manually, using four-bit numbers. Each multiplier bit is examined from right to left. If a bit is equal to 1, an appropriately shifted version of the multiplicand is added to form a *partial product*. If the multiplier bit is equal to 0, then nothing is added. The sum of all shifted versions of the multiplicand is the desired product. Note that the product occupies eight bits.



(a) Multiplication by hand

(b) Using multiple adders


m_3 m_2 m_1 m_0
× q_3 q_2 q_1 q_0
Partial product 0 m_3q_0 m_2q_0 m_1q_0 m_0q_0
+ m_3q_1 m_2q_1 m_1q_1 m_0q_1
Partial product 1 $PP1_5$ $PP1_4$ $PP1_3$ $PP1_2$ $PP1_1$
+ m_3q_2 m_2q_2 m_1q_2 m_0q_2
Partial product 2 $PP2_6$ $PP2_5$ $PP2_4$ $PP2_3$ $PP2_2$
+ m_3q_3 m_2q_3 m_1q_3 m_0q_3
Product P p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

(c) Hardware implementation

Figure 3.27 Multiplication of unsigned numbers.

3.6.1 ARRAY MULTIPLIER FOR UNSIGNED NUMBERS

Figure 3.27*b* indicates how multiplication may be performed by using multiple adders. In each step a four-bit adder is used to compute the new partial product. Note that as the computation progresses, the least-significant bits are not affected by subsequent additions; hence they can be passed directly to the final product, as indicated by red arrows. Of course, these bits are a part of the partial products as well.

The same scheme can be used to design a multiplier circuit. We will stay with four-bit numbers to keep the discussion simple. Let the multiplicand, multiplier, and product be denoted as $M = m_3m_2m_1m_0$, $Q = q_3q_2q_1q_0$, and $P = p_7p_6p_5p_4p_3p_2p_1p_0$, respectively. Figure 3.27*c* shows the required operations. Partial product 0 is obtained by using the AND of q_0 with each bit of M, which produces 0 if $q_0 = 0$ and M if $q_0 = 1$. Thus

$$PP0 = m_3q_0 \;\; m_2q_0 \;\; m_1q_0 \;\; m_0q_0$$

Partial product 1, $PP1$, is generated by adding $PP0$ to a shifted version of M that is ANDed with q_1. Similarly, partial product $PP2$ is generated using the AND of q_2 with M shifted by another bit position and adding to $PP1$, and so on.

A circuit that implements the preceding operations is arranged in an array, as shown in Figure 3.28. In this figure the AND gates and full-adders that produce the partial products are shown in red, in the same way as the red highlighted rows in Figure 3.27*c*. The full-adders are connected to form ripple-carry adders. It is possible to design faster multipliers by using other types of adders.

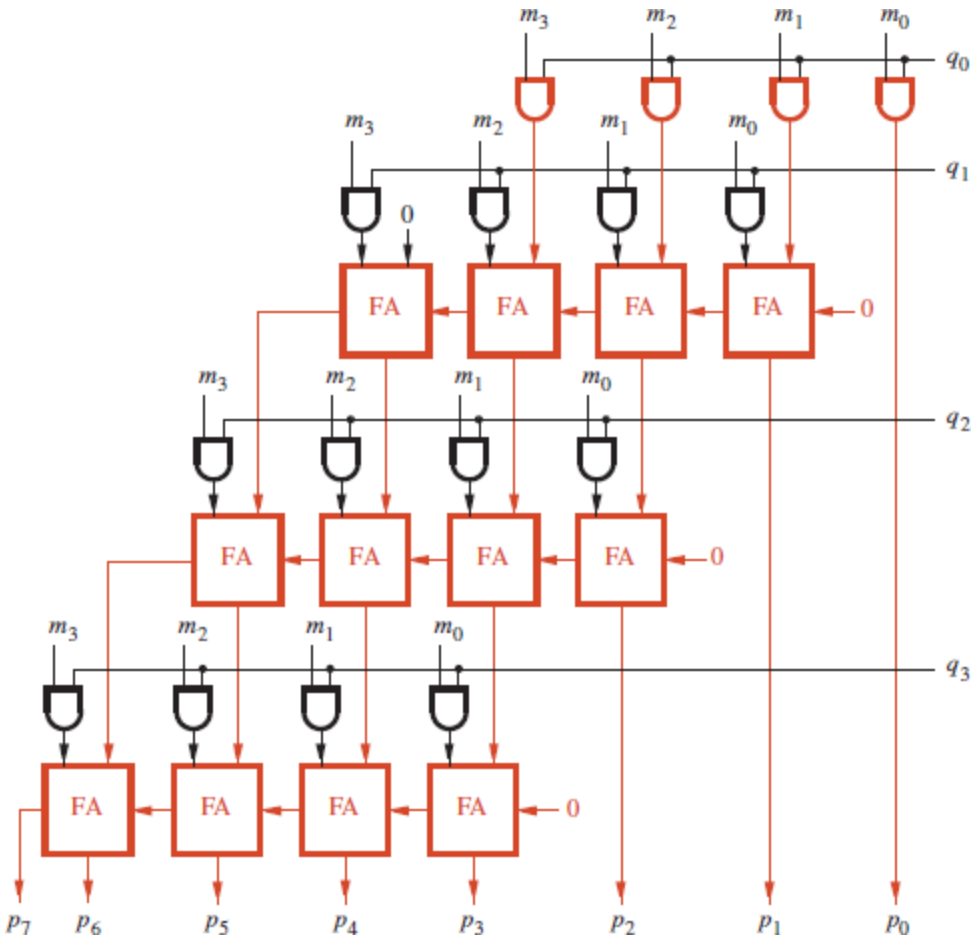


Figure 3.28 A 4 × 4 multiplier circuit.

3.6.2 MULTIPLICATION OF SIGNED NUMBERS

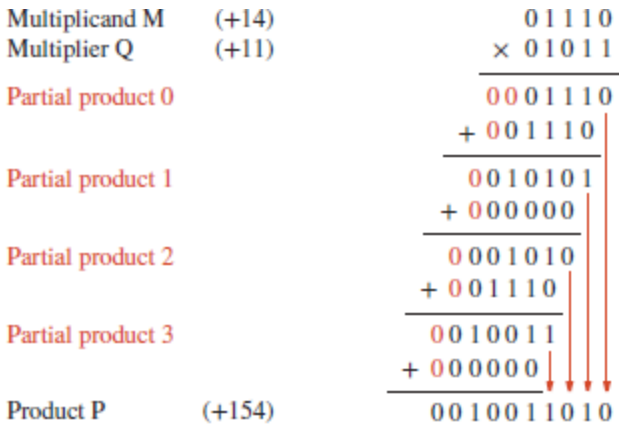
Multiplication of unsigned numbers illustrates the main issues involved in the design of multiplier circuits. Multiplication of signed numbers is somewhat more complex.

If the multiplier operand is positive, it is possible to use essentially the same scheme as for unsigned numbers. For each bit of the multiplier operand that is equal to 1, a properly shifted version of the multiplicand must be added to the partial product. The multiplicand can be either positive or negative.

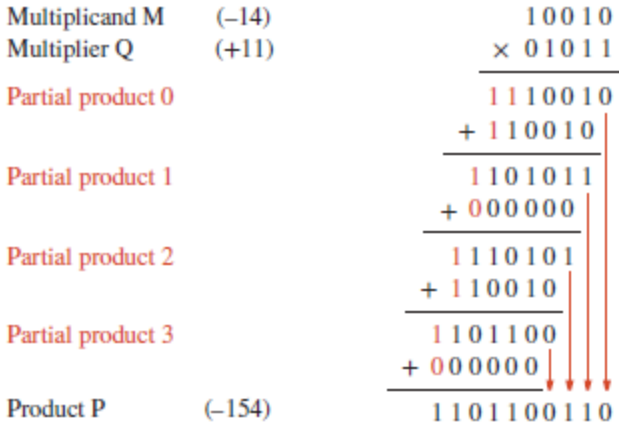
Since shifted versions of the multiplicand are added to the partial products, it is important to ensure that the numbers involved are represented correctly. For example, if the two rightmost bits of the multiplier are both equal to 1, then the first addition must produce the partial product $PP1 = M + 2M$, where M is the multiplicand. If $M = m_{n-1}m_{n-2} \cdots m_1 m_0$, then

$PP1 = m_{n-1}m_{n-2} \cdots m_1 m_0 + m_{n-1}m_{n-2} \cdots m_1 m_0 0$. The adder that performs this addition comprises circuitry that adds two operands of equal length. Since shifting the multiplicand to the left, to generate 2*M*, results in one of the operands having $n + 1$ bits, the required addition has to be performed using the second operand, *M*, represented also as an $(n + 1)$-bit number. An *n*-bit signed number is represented as an $(n + 1)$-bit number by using sign extension, that is, by replicating the sign bit as the new leftmost bit. Thus $M = m_{n-1}m_{n-2} \cdots m_1 m_0$ is represented using $(n + 1)$ bits as $M = m_{n-1}m_{n-1}m_{n-2} \cdots m_1 m_0$.}

When a shifted version of the multiplicand is added to a partial product, overflow has to be avoided. Hence the new partial product must be larger by one extra bit. Figure 3.29*a* illustrates the process of multiplying two positive numbers. The sign-extended bits are shown in red. Part (*b*) of the figure involves a negative multiplicand. Note that the resulting product has 2*n* bits in both cases.



(a) Positive multiplicand



(b) Negative multiplicand

Figure 3.29 Multiplication of signed numbers.

For a negative multiplier operand, it is possible to convert both the multiplier and the multiplicand into their 2's complements because this will not change the value of the result. Then the scheme for a positive multiplier can be used.

We have presented a relatively simple scheme for multiplication of signed numbers. There exist other techniques that are more efficient but also more complex. We will not pursue these techniques, but an interested reader may consult a variety of specialized texts.

We have discussed circuits that perform addition, subtraction, and multiplication. Another arithmetic operation that is needed in computer systems is division. Circuits that perform division are more complex; we will present an example in Chapter 7. Various techniques for performing

division are usually discussed in books on the subject of computer organization.

3.7 Other Number Representations

In the previous sections we dealt with binary integers represented in the positional number representation. Other types of numbers are also used in digital systems. In this section we will discuss briefly three other types: fixed-point, floating-point, and binary-coded decimal numbers.

3.7.1 Fixed-Point Numbers

A *fixed-point* number consists of integer and fraction parts. It can be written in the positional number representation as

$$B = b_{n-1}b_{n-2}\cdots b_1 b_0 . b_{-1}b_{-2}\cdots b_{-k}$$

The value of the number is

$$V(B) = \sum_{i=-k}^{n-1} b_i \times 2^i$$

The position of the radix point is assumed to be fixed; hence the name fixed-point number. If the radix point is not shown, then it is assumed to be to the right of the least-significant digit, which means that the number is an integer.

Logic circuits that deal with fixed-point numbers are essentially the same as those used for integers. We will not discuss them separately.

3.7.2 Floating-Point Numbers

Fixed-point numbers have a range that is limited by the significant digits used to represent the number. For example, if we use eight digits and a sign to represent decimal integers, then the range of values that can be represented is 0 to ±99999999. If eight digits are used to represent a fraction, then the representable range is 0.00000001 to ±0.99999999. In scientific applications it is often necessary to deal with numbers that are

very large or very small. Instead of using the fixed-point representation, which would require many significant digits, it is better to use the floating-point representation in which numbers are represented by a *mantissa* comprising the significant digits and an *exponent* of the radix *R*. The format is

$$Mantissa \times R^{Exponent}$$

The numbers are often *normalized*, such that the radix point is placed to the right of the first nonzero digit, as in 5.234×10^{43} or 6.31×10^{-28}.

Binary floating-point representation has been standardized by the Institute of Electrical and Electronics Engineers (IEEE) [1]. Two sizes of formats are specified in this standard—a *single-precision* 32-bit format and a *double-precision* 64-bit format. Both formats are illustrated in Figure 3.30.

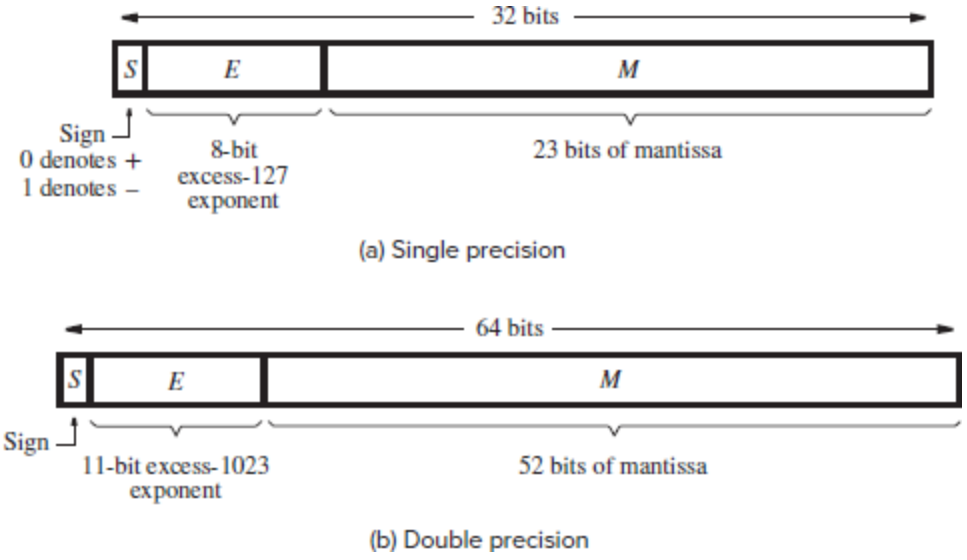


Figure 3.30 IEEE Standard floating-point formats.

Single-Precision Floating-Point Format

Figure 3.30*a* depicts the single-precision format. The leftmost bit is the sign bit—0 for positive and 1 for negative numbers. There is an 8-bit exponent field, *E*, and a 23-bit mantissa field, *M*. The exponent is with respect to the radix 2. Because it is necessary to be able to represent both very large and very small numbers, the exponent can be either positive or

negative. Instead of simply using an 8-bit signed number as the exponent, which would allow exponent values in the range -128 to 127, the IEEE standard specifies the exponent in the *excess-127* format. In this format the value 127 is added to the value of the actual exponent so that

$$Exponent = E - 127$$

In this way, E becomes a positive integer. This format is convenient for adding and subtracting floating-point numbers because the first step in these operations involves comparing the exponents to determine whether the mantissas must be appropriately shifted to add/subtract the significant bits. The range of E is 0 to 255. The extreme values of E=0 and E=255 are taken to denote the exact zero and infinity, respectively. Therefore, the normal range of the exponent is -126 to 127, which is represented by the values of E from 1 to 254.

The mantissa is represented using 23 bits. The IEEE standard calls for a normalized mantissa, which means that the most-significant bit is always equal to 1. Thus it is not necessary to include this bit explicitly in the mantissa field. Therefore, if M is the bit vector in the mantissa field, the actual value of the mantissa is 1.M, which gives a 24-bit mantissa. Consequently, the floating-point format in Figure 3.30*a* represents the number

$$\text{Value} = \pm 1.M \times 2^{E-127}$$

The size of the mantissa field allows the representation of numbers that have the precision of about seven decimal digits. The exponent field range of 2^{-126} to 2^{127} corresponds to about $10^{\pm 38}$.

Double-Precision Floating-Point Format

Figure 3.30*b* shows the double-precision format, which uses 64 bits. Both the exponent and mantissa fields are larger. This format allows greater range and precision of numbers. The exponent field has 11 bits, and it specifies the exponent in the *excess-1023* format, where

$$Exponent = E - 1023$$

The range of E is 0 to 2047, but again the values E=0 and E=2047 are used to indicate the exact zero and infinity, respectively. Thus the normal range of the exponent is -1022 to 1023, which is represented by the values of E from 1 to 2046.

The mantissa field has 52 bits. Since the mantissa is assumed to be normalized, its actual value is again 1.M. Therefore, the value of a floating-point number is

$$\text{Value} = \pm 1.M \times 2^{E-1023}$$

This format allows representation of numbers that have the precision of about 16 decimal digits and the range of approximately $10^{\pm 308}$.

Arithmetic operations using floating-point operands are significantly more complex than signed integer operations. Because this is a rather specialized domain, we will not elaborate on the design of logic circuits that can perform such operations. More complete discussions of floating-point operations are available in the published literature.

3.7.3 BINARY-CODED-DECIMAL REPRESENTATION

In digital systems it is possible to represent decimal numbers simply by encoding each digit in binary form. This is called the *binary-coded-decimal (BCD)* representation. Because there are 10 digits to encode, it is necessary to use four bits per digit. Each digit is encoded by the binary pattern that represents its unsigned value, as shown in Table 3.3. Note that only 10 of the 16 available patterns are used in BCD, which means that the remaining 6 patterns should not occur in logic circuits that operate on BCD operands; these patterns are usually treated as don't-care conditions in the design process. BCD representation was used in some early computers as well as in many handheld calculators. Its main virtue is that it provides a format that is convenient when numerical information is to be displayed on a simple digit-oriented display. Its drawbacks are complexity of circuits that perform arithmetic operations and the fact that six of the possible code patterns are wasted.

Table 3-2 Binary-coded decimal digits.

Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Even though the importance of BCD representation has diminished, it is still encountered. To give the reader an indication of the complexity of the required circuits, we will consider BCD addition in some detail.

BCD Addition

The addition of two BCD digits is complicated by the fact that the sum may exceed 9, in which case a correction will have to be made. Let $X = x_3x_2x_1x_0$ and $Y = y_3y_2y_1y_0$ represent the two BCD digits and let $S = s_3s_2s_1s_0$ be the desired sum digit, $S = X + Y$. Obviously, if $X + Y \leq 9$, then the addition is the same as the addition of 2 four-bit unsigned binary numbers. But, if $X + Y > 9$, then the result requires two BCD digits. Moreover, the four-bit sum obtained from the four-bit adder may be incorrect.

There are two cases where some correction has to be made: when the sum is greater than 9 but no carry-out is generated using four bits, and when the sum is greater than 15 so that a carry-out is generated using four bits. Figure 3.31 illustrates these cases. In the first case the four-bit addition yields $Z = 7 + 5 = 12$. To obtain a correct BCD result, we must generate $S = 2$ and a carry-out of 1. The necessary correction is apparent from the fact that the four-bit addition is a modulo-16 scheme, whereas decimal addition is a modulo-10 scheme. Therefore, a correct decimal digit can be generated by adding 6 to the result of four-bit addition whenever this result exceeds 9. Thus we can arrange the computation as follows

$$Z = X + Y$$

if $Z \leq 9$, then $S = Z$ and carry-out $= 0$

if $Z > 9$, then $S = Z + 6$ and carry-out $= 1$

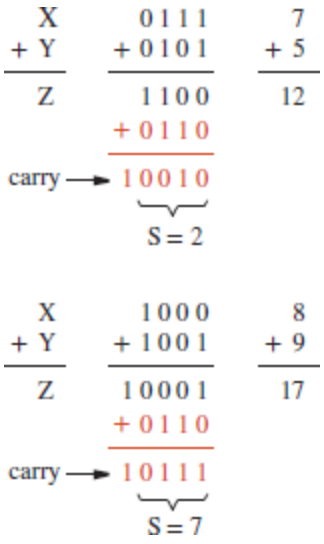


Figure 3.31 Addition of BCD digits.

The second example in Figure 3.31 shows what happens when $X + Y > 15$. In this case $Z = 17$, which is a five-bit binary number. Note that the four least-significant bits, $Z_{3\text{-}0}$, represent the value 1, while Z_4 represents the value 16. To generate a correct BCD result, it is again necessary to add 6 to the intermediate sum $Z_{3\text{-}0}$ to produce the value 7. The bit Z_4 is the carry to the next digit position.

Figure 3.32 gives a block diagram of a one-digit BCD adder that is based on this scheme. The block that detects whether $Z > 9$ produces an output signal, Adjust, which controls the multiplexer that provides the correction when needed. A second four-bit adder generates the corrected sum bits. Since whenever the adjustment of 6 is made there is a carry to the next digit position, c_{out} is just equal to the Adjust signal.

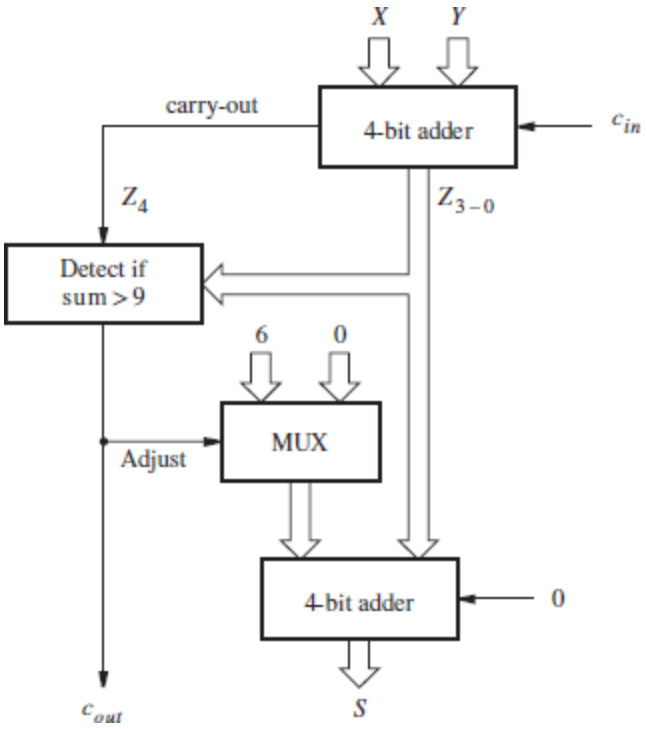


Figure 3.32 Block diagram for a one-digit BCD adder.

An implementation of this block diagram using VHDL code is shown in Figure 3.33. Inputs *X* and *Y* are defined as four-bit numbers. The sum output, *S*, is defined as a five-bit number, which allows for the carry-out to appear in bit S_4, while the sum is produced in bits $S_{3\text{-}0}$. The intermediate sum *Z* is also defined as a five-bit number. We have concatenated a 0 to input *X* in the expression Z <= (‘0’ & X) + Y because the VHDL compiler requires at least one of the operands of an arithmetic operation to have the same number of bits as in the result.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY BCD IS
    PORT ( X, Y  : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT   STD_LOGIC_VECTOR(4 DOWNTO 0) ) ;
END BCD ;

ARCHITECTURE Behavior OF BCD IS
    SIGNAL Z : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
    SIGNAL Adjust : STD_LOGIC ;
BEGIN
    Z <= ('0' & X) + Y ;
    Adjust <= '1' WHEN Z > 9 ELSE '0' ;
    S <= Z WHEN (Adjust = '0') ELSE Z + 6 ;
END Behavior ;
```

Figure 3.33 VHDL code for a one-digit BCD adder.

The statement

```
Adjust <= '1' WHEN Z > 9 ELSE '0' ;
```

uses a type of VHDL signal assignment statement that we have not seen before. It is called a *conditional signal assignment* and is used to assign one of multiple values to a signal, based on some criterion. In this case the criterion is the condition Z > 9. If this condition is satisfied, the statement assigns 1 to Adjust; otherwise, it assigns 0 to Adjust. Other examples of the conditional signal assignment are given in Chapter 4.

We should also note that we have included the *Adjust* signal in the VHDL code only to be consistent with Figure 3.32. We could just as easily have eliminated the *Adjust* signal and written the expression as

```
S <= Z WHEN Z < 10 ELSE Z + 6 ;
```

If we wish to derive a circuit to implement the block diagram in Figure 3.32 by hand, instead of by using VHDL, then the following approach can be used. To define the *Adjust* function, we can observe that the intermediate sum will exceed 9 if the carry-out from the four-bit adder is equal to 1, or if $z_3 = 1$ and either z_2 or z_1 (or both) are equal to 1. Hence the logic expression for this function is

$$\text{Adjust} = \text{Carry-out} + z_3(z_2 + z_1)$$

Instead of implementing another complete four-bit adder to perform the correction, we can use a simpler circuit because the addition of constant 6 does not require the full capability of a four-bit adder. Note that the least-significant bit of the sum, s_0, is not affected at all; hence $s_0 = z_0$. A two-bit adder may be used to develop bits s_2 and s_1. Bit s_3 is the same as z_3 if the carry-out from the two-bit adder is 0, and it is equal to $\bar{z}_3$ if this carry-out is equal to 1. A complete circuit that implements this scheme is shown in Figure 3.34. Using the one-digit BCD adder as a basic block, it is possible to build larger BCD adders in the same way as a binary full-adder is used to build larger ripple-carry binary adders.

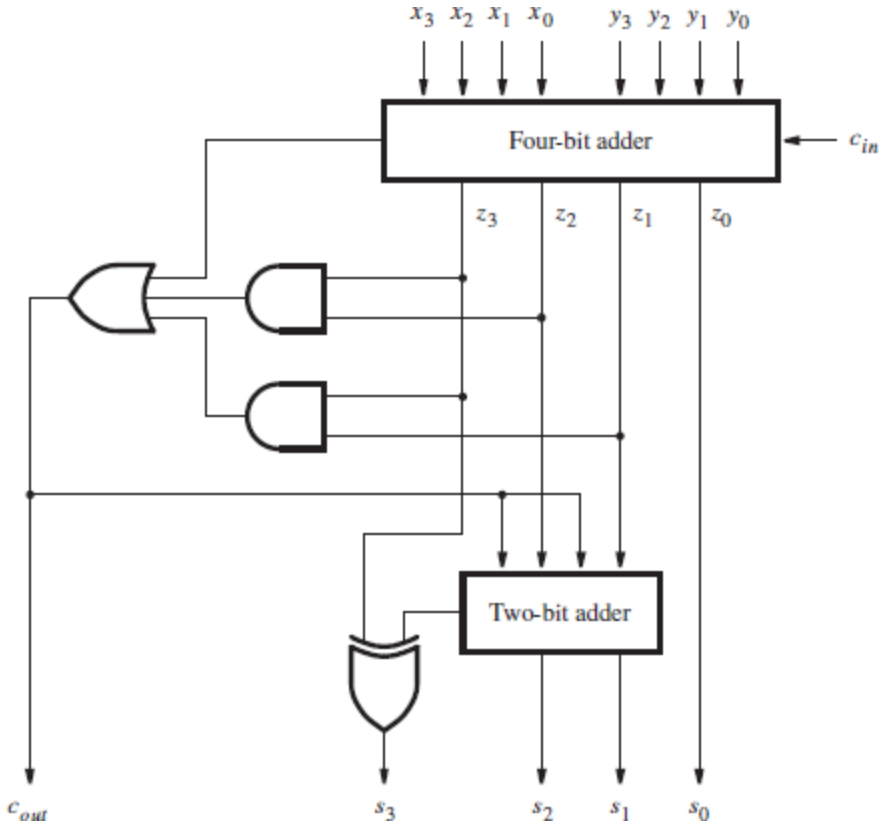


Figure 3.34 Circuit for a one-digit BCD adder.

Subtraction of BCD numbers can be handled with the radix-complement approach. Just as we use 2's complement representation to deal with negative binary numbers, we can use 10's complement representation to

deal with decimal numbers. We leave the development of such a scheme as an exercise for the reader (see Problem 3.19).

3.8 Examples of Solved Problems

This section presents some typical problems that the reader may encounter and shows how such problems can be solved.

Example 3.6

Problem: Convert the decimal number 14959 into a hexadecimal number.

Solution: An integer is converted into the hexadecimal representation by successive divisions by 16, such that in each step the remainder is a hex digit. To see why this is true, consider a four-digit number $H = h_3h_2h_1h_0$. Its value is

$$V = h_3 \times 16^3 + h_2 \times 16^2 + h_1 \times 16 + h_0$$

If we divide this by 16, we obtain

$$\frac{V}{16} = h_3 \times 16^2 + h_2 \times 16 + h_1 + \frac{h_0}{16}$$

Thus, the remainder gives h_0. Figure 3.35 shows the steps needed to perform the conversion $(14959)_{10} = (3A6F)_{16}$.

Convert $(14959)_{10}$

			Remainder	Hex digit	
14959 ÷ 16	=	934	15	F	LSB
934 ÷ 16	=	58	6	6	
58 ÷ 16	=	3	10	A	
3 ÷ 16	=	0	3	3	MSB

Result is $(3A6F)_{16}$

Figure 3.35 Conversion from decimal to hexadecimal.

Example 3.7

Problem: Convert the decimal fraction 0.8254 into binary representation.

Solution: As indicated in Section 3.7.1, a binary fraction is represented as the bit pattern $B = 0.b_{-1}b_{-2}\cdots b_{-m}$ and its value is

$$V = b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-m} \times 2^{-m}$$

Multiplying this expression by 2 gives

$$b_{-1} + b_{-2} \times 2^{-1} + \cdots + b_{-m} \times 2^{-(m-1)}$$

Here, the leftmost term is the first bit to the right of the radix point. The remaining terms constitute another binary fraction which can be manipulated in the same way. Therefore, to convert a decimal fraction into a binary fraction, we multiply the decimal number by 2 and set the computed bit to 0 if the product is less than 1 and set it to 1 if the product is greater than or equal to 1. We repeat this calculation until a sufficient number of bits are obtained to meet the desired accuracy. Note that it may not be possible to represent a decimal fraction with a binary fraction that has exactly the same value. Figure 3.36 shows the required computation that yields $(0.8254)_{10} = (0.11010011\ \ldots)_2$.

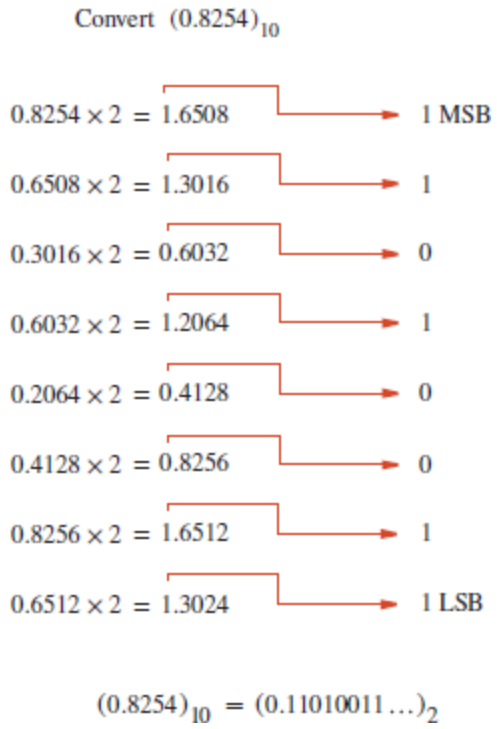


Figure 3.36 Conversion of fractions from decimal to binary.

Example 3.8

Problem: Convert the decimal fixed point number 214.45 into a binary fixed point number.

Solution: For the integer part perform successive division by 2 as illustrated in Figure 1.6. For the fractional part perform successive multiplication by 2 as described in Example 3.7. The complete computation is presented in Figure 3.37, producing $(214.45)_{10} = (11010110.0111001\ldots)_2$.

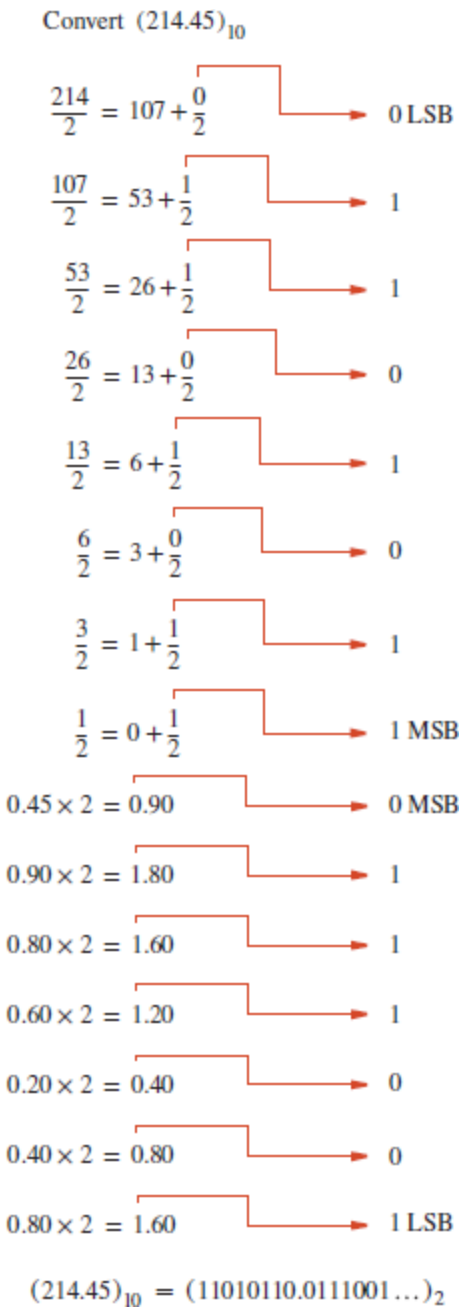


Figure 3.37 Conversion of fixed point numbers from decimal to binary.

Example 3.9

Problem: In computer computations it is often necessary to compare numbers. Two four-bit signed numbers, $X = x_3x_2x_1x_0$ and $Y = y_3y_2y_1y_0$, can

be compared by using the comparator circuit in Figure 3.38, which performs the operation X - Y. The three outputs denote the following:

- $Z = 1$ if the result is 0; otherwise $Z = 0$
- $N = 1$ if the result is negative; otherwise $N = 0$
- $V = 1$ if arithmetic overflow occurs; otherwise $V = 0$

Show how Z, N, and V can be used to determine the cases $X = Y$, $X < Y$, $X \leq Y$, $X > Y$, and $X \geq Y$.

Solution: Consider first the case $X < Y$, where the following possibilities may arise:

- If X and Y have the same sign there will be no overflow, hence $V = 0$. Then for both positive and negative X and Y the difference will be negative ($N = 1$).
- If X is negative and Y is positive, the difference will be negative ($N = 1$) if there is no overflow ($V = 0$); but the result will be positive ($N = 0$) if there is overflow ($V = 1$).

Therefore, if $X < Y$ then $N \oplus V = 1$.

The case $X = Y$ is detected by $Z = 1$. Then, $X \leq Y$ is detected by $Z + (N \oplus V) = 1$. The last two cases are just simple inverses: $X > Y$ if $\overline{Z + (N \oplus V)} = 1$ and $X \geq Y$ if $\overline{N \oplus V} = 1$.

Example 3.10

Problem: Write VHDL code to specify the circuit in Figure 3.38.

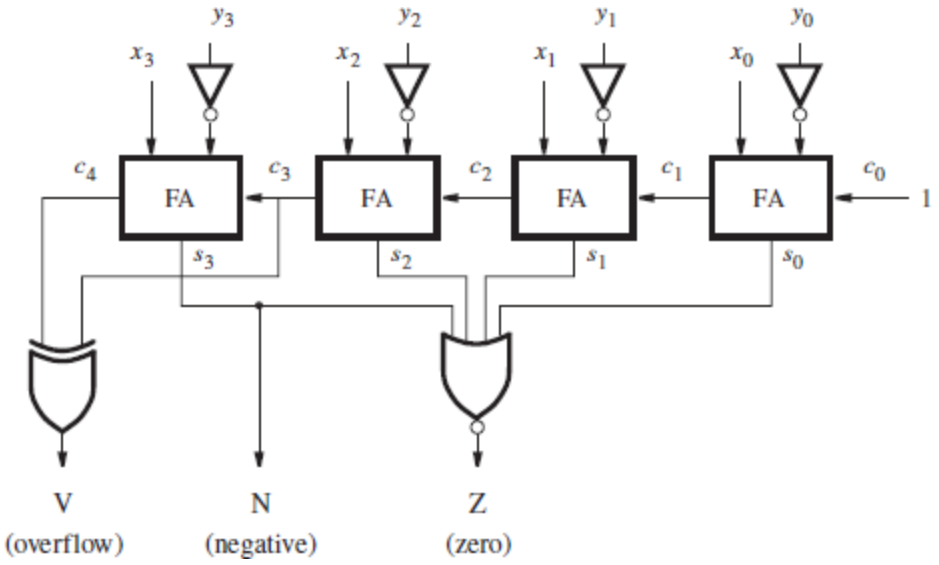


Figure 3.38 A comparator circuit.

Solution: We can specify the circuit using the structural approach as indicated in Figure 3.39. The full-adder component is defined in a package in Figure 3.20.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY comparator IS
    PORT ( X, Y     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           V, N, Z  : OUT   STD_LOGIC ) ;
END comparator ;

ARCHITECTURE Structure OF comparator IS
    SIGNAL Yn, S : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 4) ;
BEGIN
    Yn <= NOT Y;
    stage0: fulladd PORT MAP ( '1', X(0), Yn(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Yn(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Yn(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), Yn(3), S(3), C(4) ) ;
    V <= C(4) XOR C(3) ;
    N <= S(3) ;
    Z <= '1' WHEN S(3 DOWNTO 0) = "0000" ELSE '0' ;
END Structure ;
```

Figure 3.39 Structural VHDL code for the comparator circuit.

The structural approach becomes awkward when large circuits are involved, as would be the case if the comparator had 32-bit operands. An alternative is to use a behavioral specification, as shown in Figure 3.40. Note that we specified directly that *Y* should be subtracted from *X*, so that we don't have to complement *Y* explicitly. We have to specify the overflow signal, *V*, in terms of the *S* bits only, because the interstage carry signals are not accessible as explained in the discussion of Figure 3.24.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY comparator IS
    PORT ( X, Y     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           V, N, Z  : OUT   STD_LOGIC ) ;
END comparator ;

ARCHITECTURE Behavior OF comparator IS
    SIGNAL S : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
BEGIN
    S <= ('0' & X) – ('0' & Y) ;
    V <= S(4) XOR X(3) XOR Y(3) XOR S(3) ;
    N <= S(3) ;
    Z <= '1' WHEN S(3 DOWNTO 0) = 0 ELSE '0' ;
END Behavior ;
```

Figure 3.40 Behavioral VHDL code for the comparator circuit.

Example 3.11

Problem: Figure 3.28 depicts a four-bit multiplier circuit. Each row in this circuit consists of four full-adder (FA) blocks connected in a ripple-carry configuration. The delay caused by the carry signals rippling through the rows has a significant impact on the time needed to generate the output product. In an attempt to speed up the circuit, we may use the arrangement in Figure 3.41. Here, the carries in a given row are “saved” and included in the next row at the correct bit position. Then, in the first row the full-adders can be used to add three properly shifted bits of the multiplicand as selected by the multiplier bits. For example, in bit position 2 the three inputs are $m_2 q_0$, $m_1 q_1$, and $m_0 q_2$. In the last row it is still necessary to use the ripple-carry

adder. A circuit that consists of an array of full-adders connected in this manner is called a *carry-save* adder array.

What is the total delay of the circuit in Figure 3.41 compared to that of the circuit in Figure 3.28?

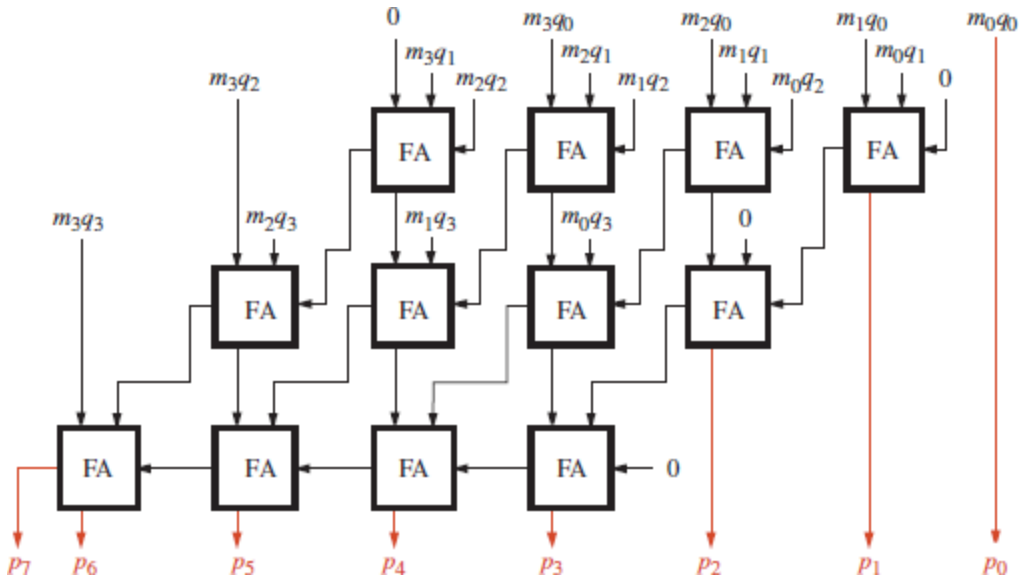


Figure 3.41 Multiplier carry-save array.

Solution: In the circuit in Figure 3.28 the longest path is through the rightmost two full-adders in the top row, followed by the two rightmost FAs in the second row, and then through all four FAs in the bottom row. Hence this delay is eight times the delay through a full-adder block. In addition, there is the AND-gate delay needed to form the inputs to the first FA in the top row. These combined delays are the critical delay, which determines the speed of the multiplier circuit.

In the circuit in Figure 3.41, the longest path is through the rightmost FAs in the first and second rows, followed by all four FAs in the bottom row. Therefore, the critical delay is six times the delay through a full-adder block plus the AND-gate delay needed to form the inputs to the first FA in the top row.

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

***3.1** Determine the decimal values of the following unsigned numbers:

(a) $(0111011110)_2$

(b) $(1011100111)_2$

(c) $(3751)_8$

(d) $(A25F)_{16}$

(e) $(F0F0)_{16}$

***3.2** Determine the decimal values of the following 1's complement numbers:

(a) 0111011110

(b) 1011100111

(c) 1111111110

***3.3** Determine the decimal values of the following 2's complement numbers:

(a) 0111011110

(b) 1011100111

(c) 1111111110

***3.4** Convert the decimal numbers 73, 1906, -95, and -1630 into signed 12-bit numbers in the following representations:

(a) Sign and magnitude

(b) 1's complement

(c) 2's complement

3.5 Perform the following operations involving four-bit 2's complement numbers and indicate whether arithmetic overflow occurs. Check your answers by converting to decimal sign-and-magnitude representation.

0110	0101	1111
+0101	+ 1110	+ 1000
0110	0101	0011
− 1011	− 0110	− 1100

3.6 Perform the following operations involving eight-bit 2's complement numbers and indicate whether arithmetic overflow occurs. Check your answers by converting to decimal sign-and-magnitude representation.

00110110	01110101	11011111
+ 01000101	+ 11011110	+ 10111000
00110110	01110101	11010011
− 00101011	− 11010110	− 11101100

3.7 Prove that the XOR operation is associative, which means that $x_i \oplus (y_i \oplus z_i) = (x_i \oplus y_i) \oplus z_i$.

3.8 Show that the circuit in Figure 3.4 implements the full-adder specified in Figure 3.3*a*.

3.9 Prove the validity of the simple rule for finding the 2's complement of a number, which was presented in Section 3.3. Recall that the rule states that scanning a number from right to left, all 0s and the first 1 are copied; then all remaining bits are complemented.

3.10 Prove the validity of the expression Overflow = $c_n \oplus c_{n-1}$ for addition of n-bit signed numbers.

3.11 Verify that a carry-out signal, c_k, from bit position k-1 of an adder circuit can be generated as $c_k = x_k \oplus y_k \oplus s_k$, where x_k and y_k are inputs and s_k is the sum bit.

***3.12** Determine the number of gates needed to implement an n-bit carry-lookahead adder, assuming no fan-in constraints. Use AND, OR, and XOR gates with any number of inputs.

***3.13** Determine the number of gates needed to implement an eight-bit carry-lookahead adder assuming that the maximum fan-in for the gates is four.

3.14 In Figure 3.17 we presented the structure of a hierarchical carry-lookahead adder. Show the complete circuit for a four-bit version of this adder, built using 2 two-bit blocks.

3.15 What is the critical delay path in the multiplier in Figure 3.28? What is the delay along this path in terms of the number of gates?

3.16 Write a VHDL module to describe the 4 × 4 multiplier shown in Figure 3.28. Synthesize a circuit from the code and verify its functional correctness.

***3.17** Consider the VHDL code in Figure P3.1. Given the relationship between the signals Input and Output, what is the functionality of the circuit described by the code? Comment on whether or not this code represents a good style to use for the functionality that it represents.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY problem IS
    PORT ( Input  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Output : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END problem ;

ARCHITECTURE LogicFunc OF problem IS
BEGIN
    WITH Input SELECT
        Output <= "0001" WHEN "0101",
                  "0010" WHEN "0110",
                  "0011" WHEN "0111",
                  "0010" WHEN "1001",
                  "0100" WHEN "1010",
                  "0110" WHEN "1011",
                  "0011" WHEN "1101",
                  "0110" WHEN "1110",
                  "1001" WHEN "1111",
                  "0000" WHEN OTHERS ;
END LogicFunc ;
```

Figure P3.1 The code for Problem 3.17.

3.18 Design a circuit that generates the 9's complement of a BCD digit. Note that the 9's complement of *d* is 9 - *d*.

3.19 Derive a scheme for performing subtraction using BCD operands. Show a block diagram for the subtractor circuit. *Hint:* Subtraction can be performed easily if the operands are in the 10's complement (radix complement) representation. In this representation the sign digit is 0 for a positive number and 9 for a negative number.

3.20 Write complete VHDL code for the circuit that you derived in Problem 3.19.

***3.21** Suppose that we want to determine how many of the bits in a three-bit unsigned number are equal to 1. Design the simplest circuit that can accomplish this task.

3.22 Repeat Problem 3.21 for a six-bit unsigned number.

3.23 Repeat Problem 3.21 for an eight-bit unsigned number.

3.24 Show a graphical interpretation of three-digit decimal numbers, similar to Figure 3.11. The leftmost digit is 0 for positive numbers and 9 for negative numbers. Verify the validity of your answer by trying a few examples of addition and subtraction.

3.25 In a ternary number system there are three digits: 0, 1, and 2. Figure P3.2 defines a ternary half-adder. Design a circuit that implements this half-adder using binary-encoded signals, such that two bits are used for each ternary digit. Let $A = a_1\ a_0$, $B = b_1\ b_0$, and $Sum = s_1\ s_0$; note that *Carry* is just a binary signal. Use the following encoding: $00 = (0)_3$, $01 = (1)_3$, and $10 = (2)_3$. Minimize the cost of the circuit.

A B	Carry	Sum
0 0	0	0
0 1	0	1
0 2	0	2
1 0	0	1
1 1	0	2
1 2	1	0
2 0	0	2
2 1	1	0
2 2	1	1

Figure P3.2 Ternary half-adder.

3.26 Design a ternary full-adder circuit, using the approach described in Problem 3.25.

3.27 Consider the subtractions 26 - 27 = 99 and 18 - 34 = 84. Using the concepts presented in Section 3.3.4, explain how these answers (99 and 84) can be interpreted as the correct signed results of these subtractions.

REFERENCES

1. Institute of Electrical and Electronic Engineers (IEEE), "A Proposed Standard for Floating-Point Arithmetic," *Computer* 14, no. 3 (March 1981), pp. 51–62.

chapter

4

COMBINATIONAL-CIRCUIT BUILDING BLOCKS

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Commonly used combinational subcircuits
- Multiplexers, which can be used for selection of signals and for implementation of general logic functions
- Circuits used for encoding, decoding, and code-conversion purposes
- Key VHDL constructs used to define combinational circuits

Previous chapters have introduced the basic techniques for design of logic circuits. In practice, a few types of logic circuits are often used as building blocks in larger designs. This chapter discusses a number of these blocks and gives examples of their use. The chapter also includes a major section on VHDL, which describes several key features of the language.

4.1 MULTIPLEXERS

Multiplexers were introduced briefly in Chapter 2. A multiplexer circuit has a number of data inputs, one or more select inputs, and one output. It passes the signal value on one of the data inputs to the output. The data input is selected by the values of the select inputs. Figure 4.1 shows a 2-to-1 multiplexer. Part (*a*) gives the symbol commonly used. The *select* input, *s*, chooses as the output of the multiplexer either input w_0 or w_1. The multiplexer's functionality can be described in the form of a truth table as shown in part (*b*) of the figure. Part (*c*) gives a sum-of-products implementation of the 2-to-1 multiplexer. Part (*d*) illustrates how it can be constructed with transmission gates which are discussed in Appendix B.

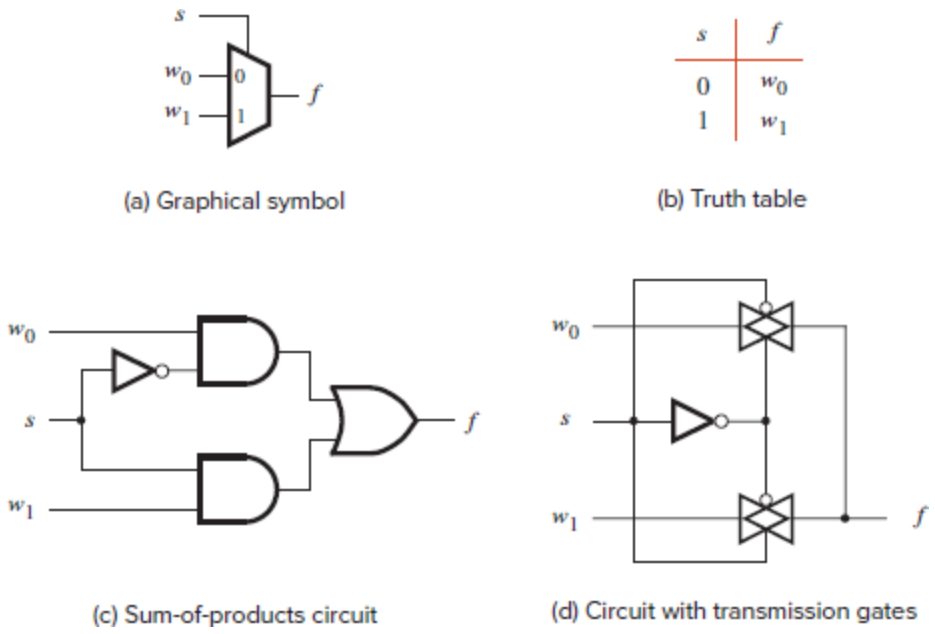


Figure 4.1 A 2-to-1 multiplexer.

Figure 4.2*a* depicts a larger multiplexer with four data inputs, w_0, …, w_3, and two select inputs, s_1 and s_0. As shown in the truth table in part (*b*) of the figure, the two-bit number represented by s_1 s_0 selects one of the data inputs as the output of the multiplexer. A sum-of-products implementation of the 4-to-1 multiplexer appears in Figure 4.2*c*. It realizes the multiplexer function

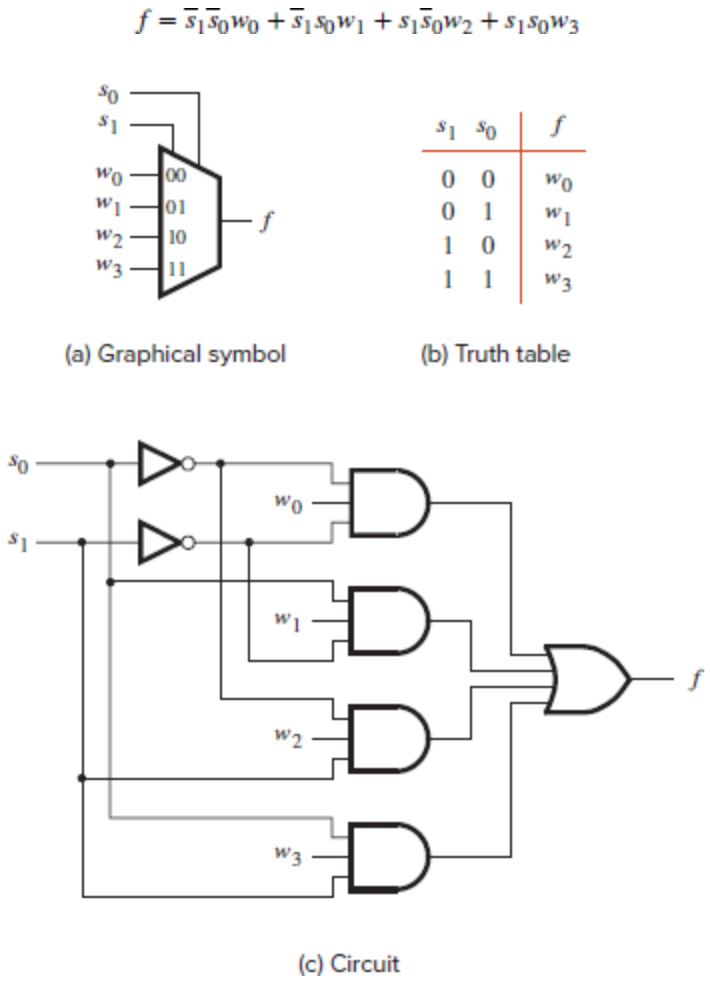


Figure 4.2 A 4-to-1 multiplexer.

It is possible to build larger multiplexers using the same approach. Usually, the number of data inputs, n, is an integer power of two. A multiplexer that has n data inputs, w_0, …, w_{n-1}, requires $\lceil \log_2 n \rceil$ select inputs. Larger multiplexers can also be constructed from smaller multiplexers. For example, the 4-to-1 multiplexer can be built using three 2-to-1 multiplexers as illustrated in Figure 4.3. Figure 4.4 shows how a 16-to-1 multiplexer is constructed with five 4-to-1 multiplexers.

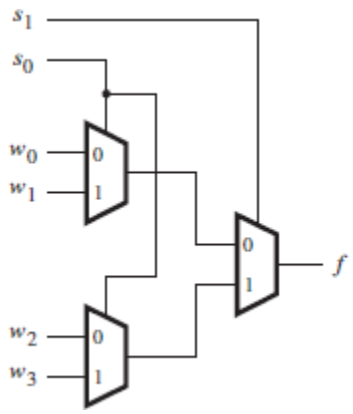


Figure 4.3 Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

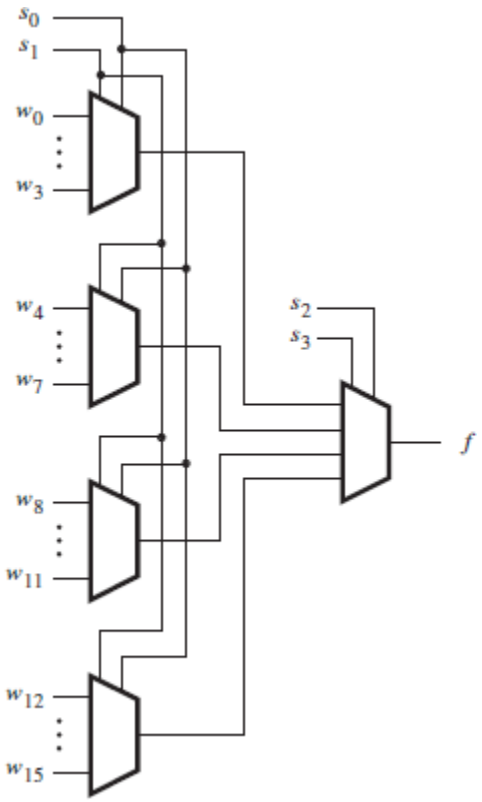


Figure 4.4 A 16-to-1 multiplexer.

Example 4.1

Figure 4.5 shows a circuit that has two inputs, x_1 and x_2, and two outputs, y_1 and y_2. As indicated by the red lines, the function of the circuit is to allow either of its inputs to be connected to either of its outputs, under the control

of another input, *s*. A circuit that has *n* inputs and *k* outputs, whose sole function is to provide a capability to connect any input to any output, is usually referred to as an *n*×*k crossbar switch*. Crossbars of various sizes can be created, with different numbers of inputs and outputs. When there are two inputs and two outputs, it is called a 2×2 crossbar.

Figure 4.5*b* shows how the 2×2 crossbar can be implemented using 2-to-1 multiplexers. The multiplexer select inputs are controlled by the signal *s*. If *s*=0, the crossbar connects x_1 to y_1 and x_2 to y_2, while if *s*=1, the crossbar connects x_1 to y_2 and x_2 to y_1. Crossbar switches are useful in many practical applications in which it is necessary to be able to connect one set of wires to another set of wires, where the connection pattern changes from time to time.

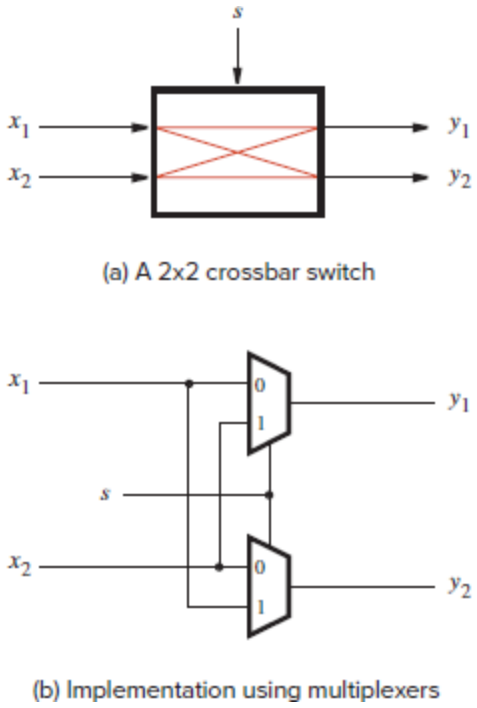
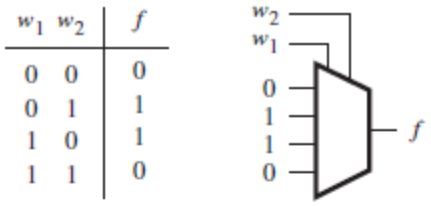


Figure 4.5 A practical application of multiplexers.

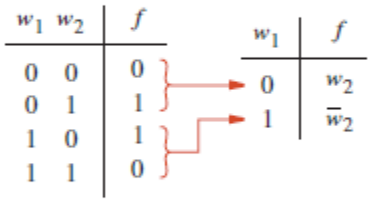
4.1.1 SYNTHESIS OF LOGIC FUNCTIONS USING MULTIPLEXERS

Multiplexers are useful in many practical applications, such as the one described above. They can also be used in a more general way to synthesize

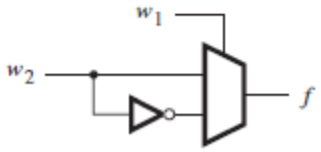
logic functions. Consider the example in Figure 4.6*a*. The truth table defines the function $f = w_1 \oplus w_2$. This function can be implemented by a 4-to-1 multiplexer in which the values of f in each row of the truth table are connected as constants to the multiplexer data inputs. The multiplexer select inputs are driven by w_1 and w_2. Thus for each valuation of w_1 w_2, the output f is equal to the function value in the corresponding row of the truth table.



(a) Implementation using a 4-to-1 multiplexer



(b) Modified truth table



(c) Circuit

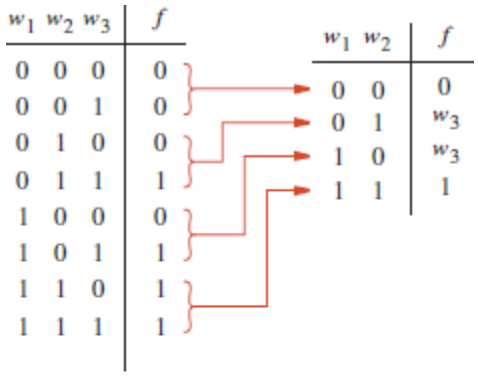
Figure 4.6 Synthesis of a logic function using multiplexers.

The above implementation is straightforward, but it is not very efficient. A better implementation can be derived by manipulating the truth table as indicated in Figure 4.6*b*, which allows f to be implemented by a single 2-to-1 multiplexer. One of the input signals, w_1 in this example, is chosen as the select input of the 2-to-1 multiplexer. The truth table is redrawn to indicate the value of f for each value of w_1. When $w_1 = 0$, f has the same value as input w_2, and when $w_1 = 1$, f has the value of $\overline{w}_2$. The circuit that implements

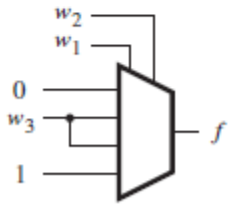
this truth table is given in Figure 4.6*c*. This procedure can be applied to synthesize a circuit that implements any logic function.

Example 4.2

Figure 4.7*a* gives the truth table for the three-input majority function, and it shows how the truth table can be modified to implement the function using a 4-to-1 multiplexer. Any two of the three inputs may be chosen as the multiplexer select inputs. We have chosen w_1 and w_2 for this purpose, resulting in the circuit in Figure 4.7*b*. Note that the majority function implements the carry-out from the full-adder introduced in Chapter 3.



(a) Modified truth table



(b) Circuit

Figure 4.7 Implementation of the three-input majority function using a 4-to-1 multiplexer.

Example 4.3

Figure 4.8*a* indicates how the function $f = w_1 \oplus w_2 \oplus w_3$ can be implemented using 2-to-1 multiplexers. When $w_1 = 0$, f is equal to the XOR

of w_2 and w_3, and when $w_1 = 1$, f is the XNOR of w_2 and w_3. Part (b) of the figure gives a corresponding circuit. The left multiplexer in the circuit produces $w_2 \oplus w_3$, using the result from Figure 4.6, and the right multiplexer uses the value of w_1 to select either $w_2 \oplus w_3$ or its complement. Note that we could have derived this circuit directly by writing the function as $f = (w_2 \oplus w_3) \oplus w_1$.

Figure 4.9 gives an implementation of the three-input XOR function using a 4-to-1 multiplexer. Choosing w_1 and w_2 for the select inputs results in the circuit shown.

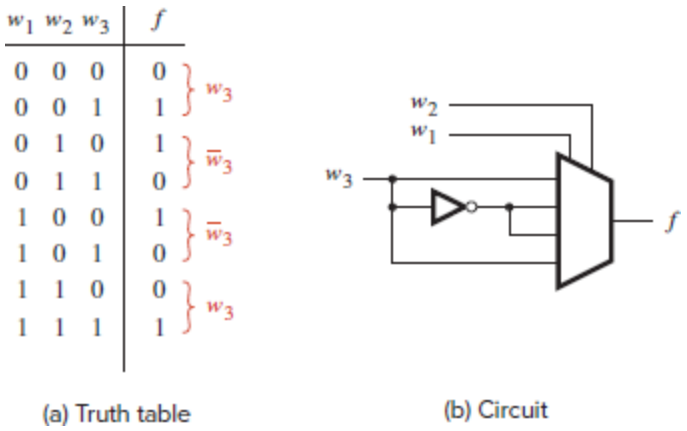


Figure 4.9 Three-input XOR implemented with a 4-to-1 multiplexer.

4.1.2 MULTIPLEXER SYNTHESIS USING SHANNON'S EXPANSION

Figures 4.6 through 4.9 illustrate how truth tables can be interpreted to implement logic functions using multiplexers. In each case the inputs to the multiplexers are the constants 0 and 1, or some variable or its complement. Besides using such simple inputs, it is possible to connect more complex circuits as inputs to a multiplexer, allowing functions to be synthesized using a combination of multiplexers and other logic gates. Suppose that we want to implement the three-input majority function in Figure 4.7 using a 2-to-1 multiplexer in this way. Figure 4.10 shows an intuitive way of realizing this function. The truth table can be modified as shown on the right. If $w_1 =$

0, then $f = w_2\, w_3$, and if $w_1 = 1$, then $f = w_2 + w_3$. Using w_1 as the select input for a 2-to-1 multiplexer leads to the circuit in Figure 4.10*b*.

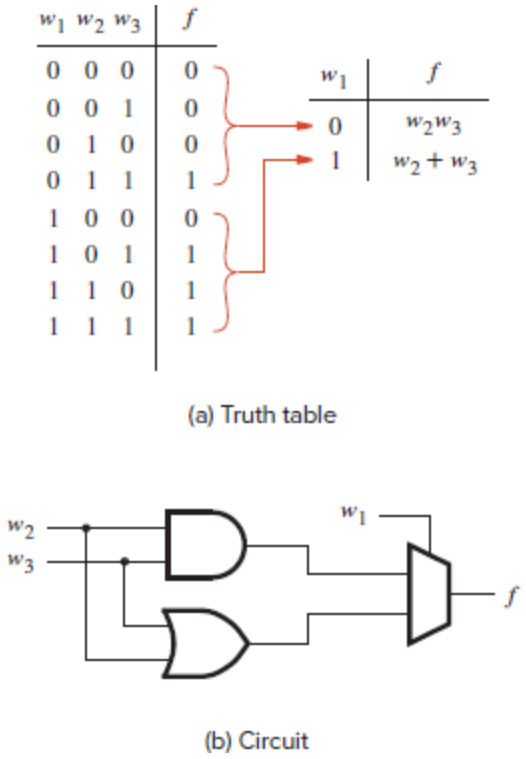


Figure 4.10 The three-input majority function implemented using a 2-to-1 multiplexer.

This implementation can be derived using algebraic manipulation as follows. The function in Figure 4.10*a* is expressed in sum-of-products form as

$$f = \overline{w}_1 w_2 w_3 + w_1 \overline{w}_2 w_3 + w_1 w_2 \overline{w}_3 + w_1 w_2 w_3$$

It can be manipulated into

$$\begin{aligned} f &= \overline{w}_1 (w_2 w_3) + w_1 (\overline{w}_2 w_3 + w_2 \overline{w}_3 + w_2 w_3) \\ &= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3) \end{aligned}$$

which corresponds to the circuit in Figure 4.10*b*.

Multiplexer implementations of logic functions require that a given function be decomposed in terms of the variables that are used as the select inputs. This can be accomplished by means of a theorem proposed by the famous mathematician Claude Shannon.

Shannon's Expansion Theorem

Any Boolean function $f(w_1, \ldots, w_n)$ can be written in the form

$$f(w_1, w_2, \ldots, w_n) = \overline{w}_1 \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$$

This expansion is done using w_1, but in general it can be done in terms of any of the n variables. We will leave the proof of the theorem as an exercise for the reader (see Problem 4.9).

To illustrate its use, we can apply the theorem to the three-input majority function, which can be written as

$$f(w_1, w_2, w_3) = w_1 w_2 + w_1 w_3 + w_2 w_3$$

Expanding this function in terms of w_1 gives

$$\begin{aligned} f &= \overline{w}_1 (0 \cdot w_2 + 0 \cdot w_3 + w_2 w_3) + w_1 (1 \cdot w_2 + 1 \cdot w_3 + w_2 w_3) \\ &= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3) \end{aligned}$$

which is the expression that we derived above.

For the three-input XOR function, we have

$$\begin{aligned} f &= w_1 \oplus w_2 \oplus w_3 \\ &= \overline{w}_1 (0 \oplus w_2 \oplus w_3) + w_1 (1 \oplus w_2 \oplus w_3) \\ &= \overline{w}_1 \cdot (w_2 \oplus w_3) + w_1 \cdot \overline{(w_2 \oplus w_3)} \end{aligned}$$

which gives the circuit in Figure 4.8*b*.

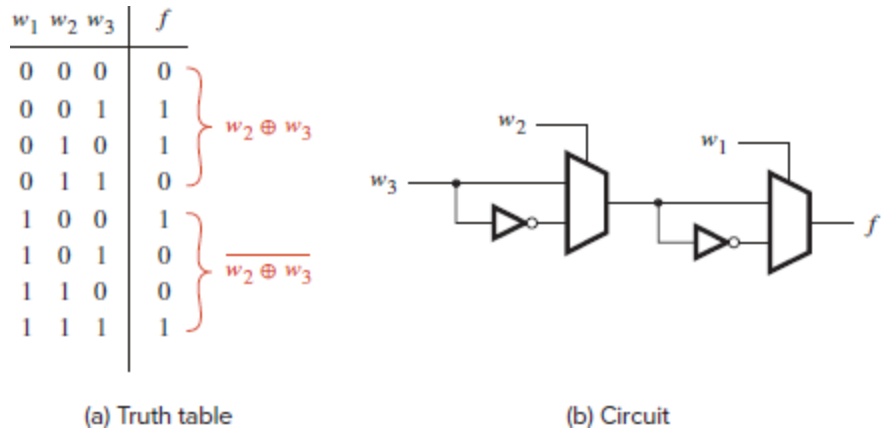


Figure 4.8 Three-input XOR implemented with 2-to-1 multiplexers.

In Shannon's expansion the term $f(0, w_2, \ldots, w_n)$ is called the *cofactor* of f with respect to $\overline{w}_1$; it is denoted in shorthand notation as $f_{\overline{w}_1}$. Similarly, the term $f(1, w_2, \ldots, w_n)$ is called the cofactor of f with respect to w_1, written f_{w_1}. Hence we can write

$$f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

In general, if the expansion is done with respect to variable w_i, then $f_{\overline{w}_i}$ denotes $f(w_1, \ldots, w_{i-1}, 0, w_{i+1}, \ldots, w_n)$, f_{w_i} denotes $f(w_1, \ldots, w_{i-1}, 1, w_{i+1}, \ldots, w_n)$, and

$$f(w_1, \ldots, w_n) = \overline{w}_i f_{\overline{w}_i} + w_i f_{w_i}$$

The complexity of the logic expression may vary depending on which variable, w_i, is used, as illustrated in Example 4.4.

Example 4.4

For the function $f = \overline{w}_1 w_3 + w_2 \overline{w}_3$, decomposition using w_1 gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1 (w_3 + w_2) + w_1 (w_2 \overline{w}_3) \end{aligned}$$

Using w_2 instead of w_1 produces

$$\begin{aligned} f &= \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2} \\ &= \overline{w}_2 (\overline{w}_1 w_3) + w_2 (\overline{w}_1 w_3 + \overline{w}_3) \\ &= \overline{w}_2 (\overline{w}_1 w_3) + w_2 (\overline{w}_1 + \overline{w}_3) \end{aligned}$$

Finally, using w_3 gives

$$\begin{aligned} f &= \overline{w}_3 f_{\overline{w}_3} + w_3 f_{w_3} \\ &= \overline{w}_3 (w_2) + w_3 (\overline{w}_1) \end{aligned}$$

The results generated using w_1 and w_2 have the same cost, but the expression produced using w_3 has a lower cost. In practice, when performing decompositions of this type it is useful to try a number of alternatives and choose the one that produces the best result.

Shannon's expansion can be done in terms of more than one variable. For example, expanding a function in terms of w_1 and w_2 gives

$$f(w_1, \ldots, w_n) = \overline{w}_1\overline{w}_2 \cdot f(0, 0, w_3, \ldots, w_n) + \overline{w}_1 w_2 \cdot f(0, 1, w_3, \ldots, w_n) + w_1\overline{w}_2 \cdot f(1, 0, w_3, \ldots, w_n) + w_1 w_2 \cdot f(1, 1, w_3, \ldots, w_n)$$

This expansion gives a form that can be implemented using a 4-to-1 multiplexer. If Shannon's expansion is done in terms of all n variables, then the result is the canonical sum-of-products form, which was defined in Section 2.6.1.

Example 4.5

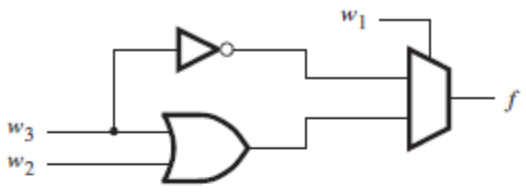
Assume that we wish to implement the function

$$f = \overline{w}_1\overline{w}_3 + w_1 w_2 + w_1 w_3$$

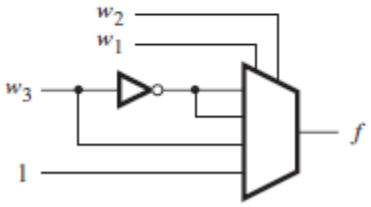
using a 2-to-1 multiplexer and any other necessary gates. Shannon's expansion using w_1 gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1(\overline{w}_3) + w_1(w_2 + w_3) \end{aligned}$$

The corresponding circuit is shown in Figure 4.11*a*. Assume now that we wish to use a 4-to-1 multiplexer instead. Further decomposition using w_2 gives



(a) Using a 2-to-1 multiplexer



(b) Using a 4-to-1 multiplexer

Figure 4.11 The circuits synthesized in Example 4.5.

$$f = \overline{w}_1\overline{w}_2 f_{\overline{w}_1\overline{w}_2} + \overline{w}_1 w_2 f_{\overline{w}_1 w_2} + w_1\overline{w}_2 f_{w_1\overline{w}_2} + w_1 w_2 f_{w_1 w_2}$$
$$= \overline{w}_1\overline{w}_2(\overline{w}_3) + \overline{w}_1 w_2(\overline{w}_3) + w_1\overline{w}_2(w_3) + w_1 w_2(1)$$

The circuit is shown in Figure 4.11*b*.

Example 4.6

Consider the three-input majority function

$$f = w_1w_2 + w_1w_3 + w_2w_3$$

We wish to implement this function using only 2-to-1 multiplexers. Shannon's expansion using w_1 yields

$$f = \overline{w}_1(w_2w_3) + w_1(w_2 + w_3 + w_2w_3)$$
$$= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3)$$

Let $g = w_2\, w_3$ and $h = w_2 + w_3$. Expansion of both g and h using w_2 gives

$$g = \overline{w}_2(0) + w_2(w_3)$$
$$h = \overline{w}_2(w_3) + w_2(1)$$

The corresponding circuit is shown in Figure 4.12. It is equivalent to the 4-to-1 multiplexer circuit derived using a truth table in Figure 4.7.

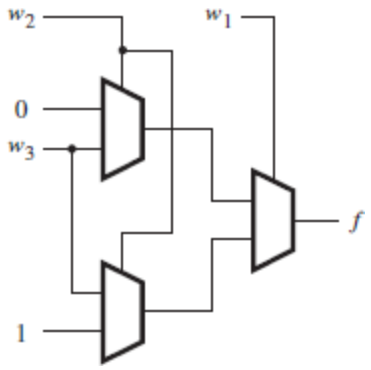


Figure 4.12 The circuit synthesized in Example 4.6.

4.2 DECODERS

Consider the logic circuit in Figure 4.13. It has two inputs, w_1 and w_0, and four outputs, y_0, y_1, y_2, and y_3. As shown in the truth table, only one of the outputs is asserted at a time, and each output corresponds to one valuation of the inputs. Setting the inputs w_1 w_0 to 00, 01, 10, or 11 causes the output y_0, y_1, y_2, or y_3 to be set to 1, respectively. This type of circuit is called a *binary decoder*. Its inputs represent a binary number, which is decoded to assert the corresponding output. A circuit symbol and logic circuit for this decoder are shown in parts (*b*) and (*c*) of the figure. Each output is driven by an AND gate that decodes the corresponding valuation of w_1 w_0.

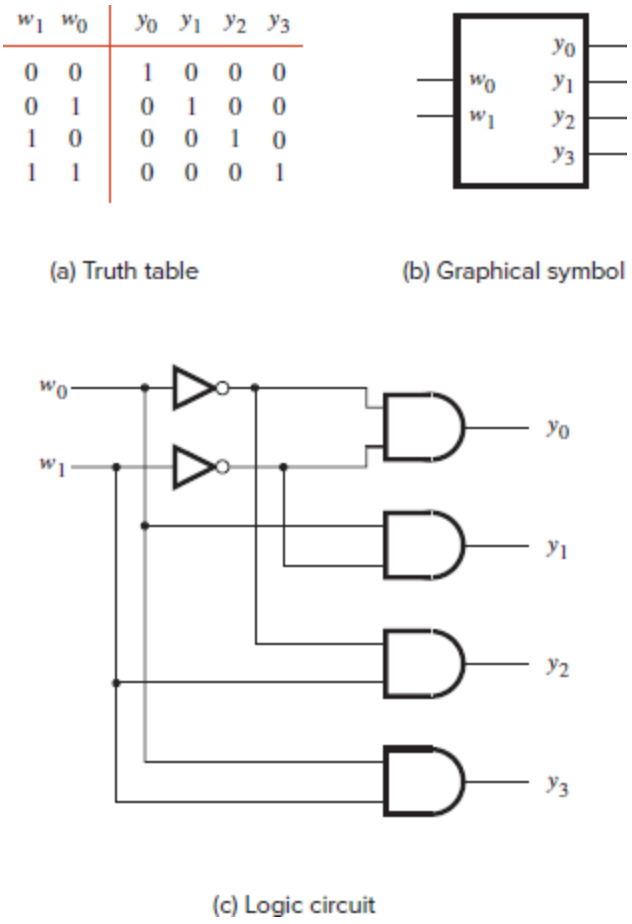


Figure 4.13 A 2-to-4 decoder.

It is useful to include an *enable* input, *En*, in a decoder circuit, as illustrated in Figure 4.14. When enabled by setting *En* = 1 the decoder behaves as presented in Figure 4.13. But, if it is disabled by setting *En* = 0, then none of the outputs are asserted. Note that only five rows are shown in the truth table, because if *En* = 0 then all outputs are equal to 0 regardless of the values of w_1 and w_0. The truth table indicates this by showing *x* when it does not matter whether the variable in question has the value 0 or 1. A graphical symbol for this decoder is given in Figure 4.14*b*. Part (*c*) of the figure shows how the enable capability can be included in the decoder of Figure 4.13*c*. A binary decoder with *n* inputs has 2^n outputs. A graphical symbol for an *n*-to-2^n decoder is shown in Figure 4.14*d*.

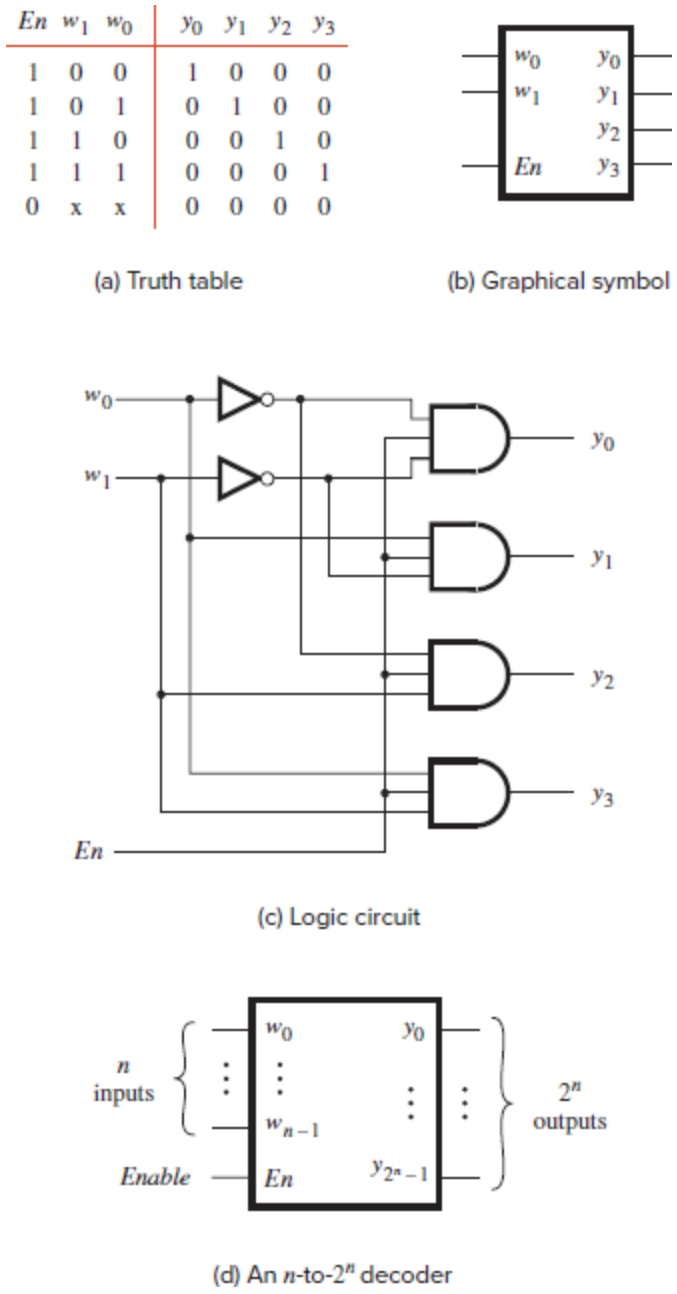


Figure 4.14 Binary decoder.

A *k*-bit binary code in which exactly one of the bits is set to 1 at a time is referred to as *one-hot encoded*, meaning that the single bit that is set to 1 is deemed to be "hot." The outputs of an enabled binary decoder are one-hot encoded.

We should also note that decoders can be designed to have either active-high or active-low outputs. In our discussion, we have assumed that active-high outputs are needed.

Larger decoders can be built using the sum-of-products structure in Figure 4.14*c*, or else they can be constructed from smaller decoders. Figure 4.15 shows how a 3-to-8 decoder is built with two 2-to-4 decoders. The w_2 input drives the enable inputs of the two decoders. The top decoder is enabled if $w_2 = 0$, and the bottom decoder is enabled if $w_2 = 1$. This concept can be applied for decoders of any size. Figure 4.16 shows how five 2-to-4 decoders can be used to construct a 4-to-16 decoder. Because of its treelike structure, this type of circuit is often referred to as a *decoder tree*.

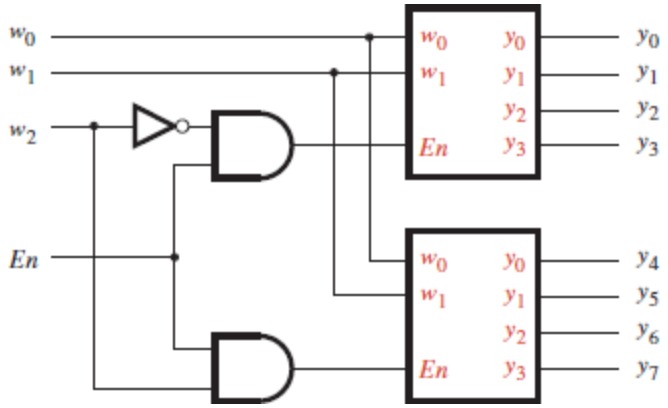


Figure 4.15 A 3-to-8 decoder using two 2-to-4 decoders.

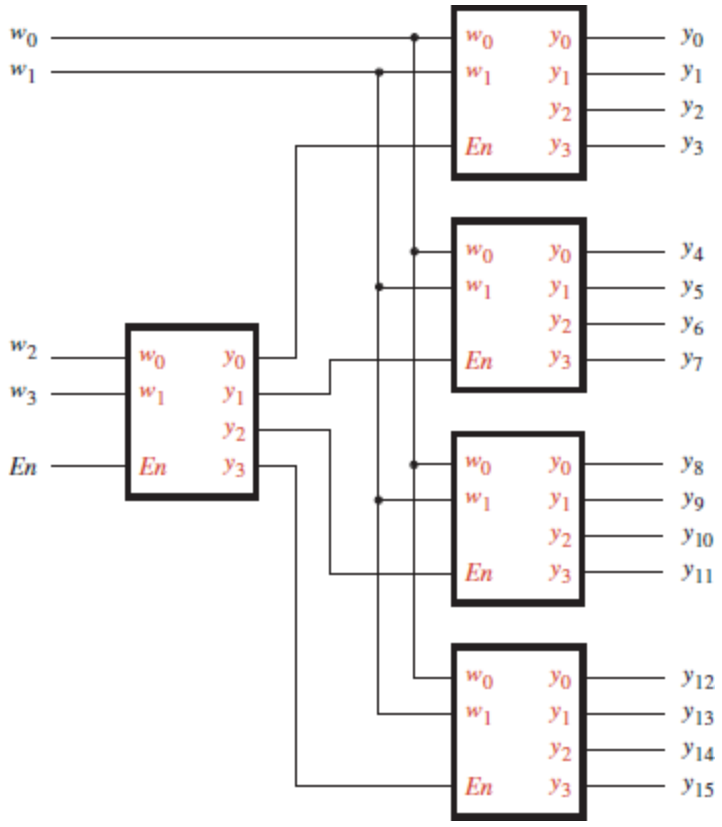


Figure 4.16 A 4-to-16 decoder built using a decoder tree.

Example 4.7

Decoders are useful for many practical purposes. In Figure 4.2*c* we showed the sum-of-products implementation of the 4-to-1 multiplexer, which requires AND gates to distinguish the four different valuations of the select inputs s_1 and s_0. Since a decoder evaluates the values on its inputs, it can be used to build a multiplexer as illustrated in Figure 4.17. The enable input of the decoder is not needed in this case, and it is set to 1. The four outputs of the decoder represent the four valuations of the select inputs.

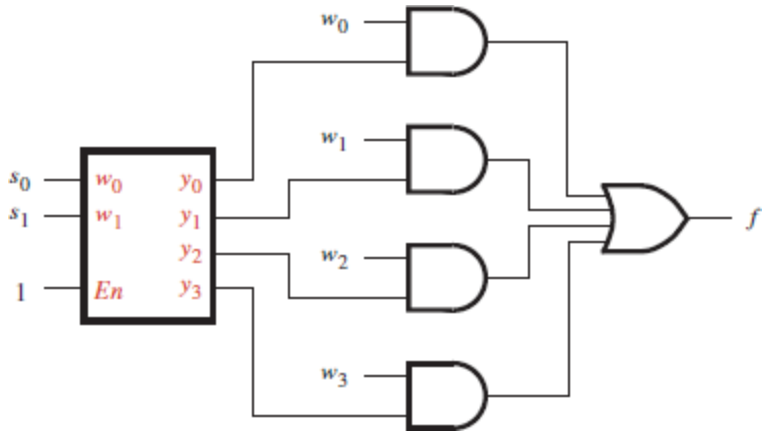


Figure 4.17 A 4-to-1 multiplexer built using a decoder.

4.2.1 DEMULTIPLEXERS

We showed in Section 4.1 that a multiplexer has one output, n data inputs, and $\lceil \log_2 n \rceil$ select inputs. The purpose of the multiplexer circuit is to *multiplex* the n data inputs onto the single data output under control of the select inputs. A circuit that performs the opposite function, namely, placing the value of a single data input onto multiple data outputs, is called a *demultiplexer*. The demultiplexer can be implemented using a decoder circuit. For example, the 2-to-4 decoder in Figure 4.14 can be used as a 1-to-4 demultiplexer. In this case the *En* input serves as the data input for the demultiplexer, and the y_0 to y_3 outputs are the data outputs. The valuation of w_1 w_0 determines which of the outputs is set to the value of *En*. To see how the circuit works, consider the truth table in Figure 4.14*a*. When $En = 0$, all the outputs are set to 0, including the one selected by the valuation of w_1 w_0. When $En = 1$, the valuation of w_1 w_0 sets the appropriate output to 1.

In general, an n-to-2^n decoder circuit can be used as a 1-to-n demultiplexer. However, in practice decoder circuits are used much more often as decoders rather than as demultiplexers.

4.3 ENCODERS

An encoder performs the opposite function of a decoder. It encodes given information into a more compact form.

4.3.1 BINARY ENCODERS

A *binary encoder* encodes information from 2^n inputs into an n-bit code, as indicated in Figure 4.18. Exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1. The truth table for a 4-to-2 encoder is provided in Figure 4.19*a*. Observe that the output y_0 is 1 when either input w_1 or w_3 is 1, and output y_1 is 1 when input w_2 or w_3 is 1. Hence these outputs can be generated by the circuit in Figure 4.19*b*. Note that we assume that the inputs are one-hot encoded. All input patterns that have multiple inputs set to 1 are not shown in the truth table, and they are treated as don't-care conditions.

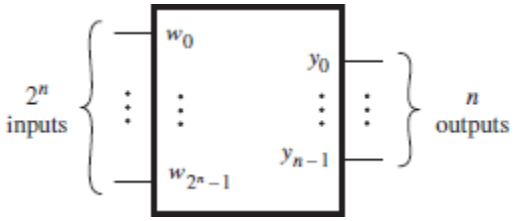
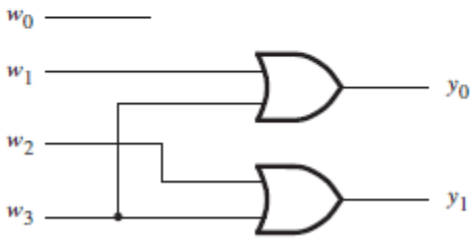


Figure 4.18 A 2^n-to-n binary encoder.

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth table



(b) Circuit

Figure 4.19 A 4-to-2 binary encoder.

Encoders are used to reduce the number of bits needed to represent given information. A practical use of encoders is for transmitting information in a digital system. Encoding the information allows the transmission link to be built using fewer wires. Encoding is also useful if information is to be stored for later use because fewer bits need to be stored.

4.3.2 PRIORITY ENCODERS

Another useful class of encoders is based on the priority of input signals. In a *priority encoder* each input has a priority level associated with it. The encoder outputs indicate the active input that has the highest priority. When an input with a high priority is asserted, the other inputs with lower priority are ignored. The truth table for a 4-to-2 priority encoder is shown in Figure 4.20. It assumes that w_0 has the lowest priority and w_3 the highest. The outputs y_1 and y_0 represent the binary number that identifies the highest priority input set to 1. Since it is possible that none of the inputs is equal to 1, an output, z, is provided to indicate this condition. It is set to 1 when at least one of the inputs is equal to 1. It is set to 0 when all inputs are equal to 0. The outputs y_1 and y_0 are not meaningful in this case, and hence the first row of the truth table can be treated as a don't-care condition for y_1 and y_0.

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Figure 4.20 Truth table for a 4-to-2 priority encoder.

The behavior of the priority encoder is most easily understood by first considering the last row in the truth table. It specifies that if input w_3 is 1, then the outputs are set to $y_1\ y_0 = 11$. Because w_3 has the highest priority level, the values of inputs w_2, w_1, and w_0 do not matter. To reflect the fact that their values are irrelevant, w_2, w_1, and w_0 are denoted by the symbol x in the truth table. The second-last row in the truth table stipulates that if w_2

= 1, then the outputs are set to $y_1\ y_0 = 10$, but only if $w_3 = 0$. Similarly, input w_1 causes the outputs to be set to $y_1\ y_0 = 01$ only if both w_3 and w_2 are 0. Input w_0 produces the outputs $y_1\ y_0 = 00$ only if w_0 is the only input that is asserted.

A logic circuit that implements the truth table can be synthesized by using the techniques developed in Chapter 2. However, a more convenient way to derive the circuit is to define a set of intermediate signals, $i_0, \ldots, i_3$, based on the observations above. Each signal, i_k, is equal to 1 only if the input with the same index, w_k, represents the highest-priority input that is set to 1. The logic expressions for $i_0, \ldots, i_3$ are

$$\begin{aligned} i_0 &= \overline{w}_3\overline{w}_2\overline{w}_1 w_0 \\ i_1 &= \overline{w}_3\overline{w}_2 w_1 \\ i_2 &= \overline{w}_3 w_2 \\ i_3 &= w_3 \end{aligned}$$

Using the intermediate signals, the rest of the circuit for the priority encoder has the same structure as the binary encoder in Figure 4.19, namely

$$\begin{aligned} y_0 &= i_1 + i_3 \\ y_1 &= i_2 + i_3 \end{aligned}$$

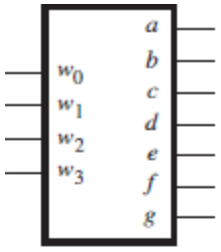
The output z is given by

$$z = i_0 + i_1 + i_2 + i_3$$

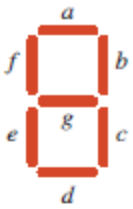
4.4 Code Converters

The purpose of the decoder and encoder circuits is to convert from one type of input encoding to a different output encoding. For example, a 3-to-8 binary decoder converts from a binary number on the input to a one-hot encoding at the output. An 8-to-3 binary encoder performs the opposite conversion. There are many other possible types of code converters. One common example is a BCD-to-7-segment decoder, which was introduced in Section 2.14. A similar decoder is often used to display hexadecimal information on seven-segment displays. As explained in Section 3.1.2, long

binary numbers are easier to deal with visually if they are represented in the hexadecimal form. A hex-to-7-segment decoder can be implemented as shown in Figure 4.21. Digits 0 to 9 are displayed the same as in the case of the BCD-to-7-segment decoder. Digits 10 to 15 are displayed as A, b, C, d, E, and F.



(a) Code converter



(b) 7-segment display

w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1	1	1
1	1	0	0	1	0	0	1	1	1	0
1	1	0	1	0	1	1	1	1	0	1
1	1	1	0	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	1	1	1

(c) Truth table

Figure 4.21 A hex-to-7-segment display code converter.

We should note that although the word *decoder* is traditionally used for such circuits, a more appropriate term is *code converter*. The term *decoder* is more appropriate for circuits that produce one-hot encoded outputs.

4.5 Arithmetic Comparison Circuits

Chapter 3 presented arithmetic circuits that perform addition, subtraction, and multiplication of binary numbers. Another useful type of arithmetic circuit compares the relative sizes of two binary numbers. Such a circuit is called a *comparator*. This section considers the design of a comparator that has two *n*-bit inputs, *A* and *B*, which represent unsigned binary numbers. The comparator produces three outputs, called *AeqB*, *AgtB*, and *AltB*. The *AeqB* output is set to 1 if *A* and *B* are equal. The *AgtB* output is 1 if *A* is greater than *B*, and the *AltB* output is 1 if *A* is less than *B*.

The desired comparator can be designed by creating a truth table that specifies the three outputs as functions of *A* and *B*. However, even for moderate values of *n*, the truth table is large. A better approach is to derive the comparator circuit by considering the bits of *A* and *B* in pairs. We can illustrate this by a small example, where $n = 4$.

Let $A = a_3a_2a_1a_0$ and $B = b_3b_2b_1b_0$. Define a set of intermediate signals called i_3,i_2,i_1, and i_0. Each signal, i_k, is 1 if the bits of *A* and *B* with the same index are equal. That is, $i_k = \overline{a_k \oplus b_k}$. The comparator's *AeqB* output is then given by

$$AeqB = i_3 i_2 i_1 i_0$$

An expression for the *AgtB* output can be derived by considering the bits of *A* and *B* in the order from the most-significant bit to the least-significant bit. The first bit-position, *k*, at which a_k and b_k differ determines whether *A* is less than or greater than *B*. If $a_k = 0$ and $b_k = 1$, then $A < B$. But if $a_k = 1$ and $b_k = 0$, then $A > B$. The *AgtB* output is defined by

$$AgtB = a_3\bar{b}_3 + i_3a_2\bar{b}_2 + i_3i_2a_1\bar{b}_1 + i_3i_2i_1a_0\bar{b}_0$$

The i_k signals ensure that only the first bits, considered from the left to the right, of *A* and *B* that differ determine the value of *AgtB*.

The *AltB* output can be derived by using the other two outputs as

$$AltB = \overline{AeqB + AgtB}$$

A logic circuit that implements the four-bit comparator circuit is shown in Figure 4.22. This approach can be used to design a comparator for any value of *n*.

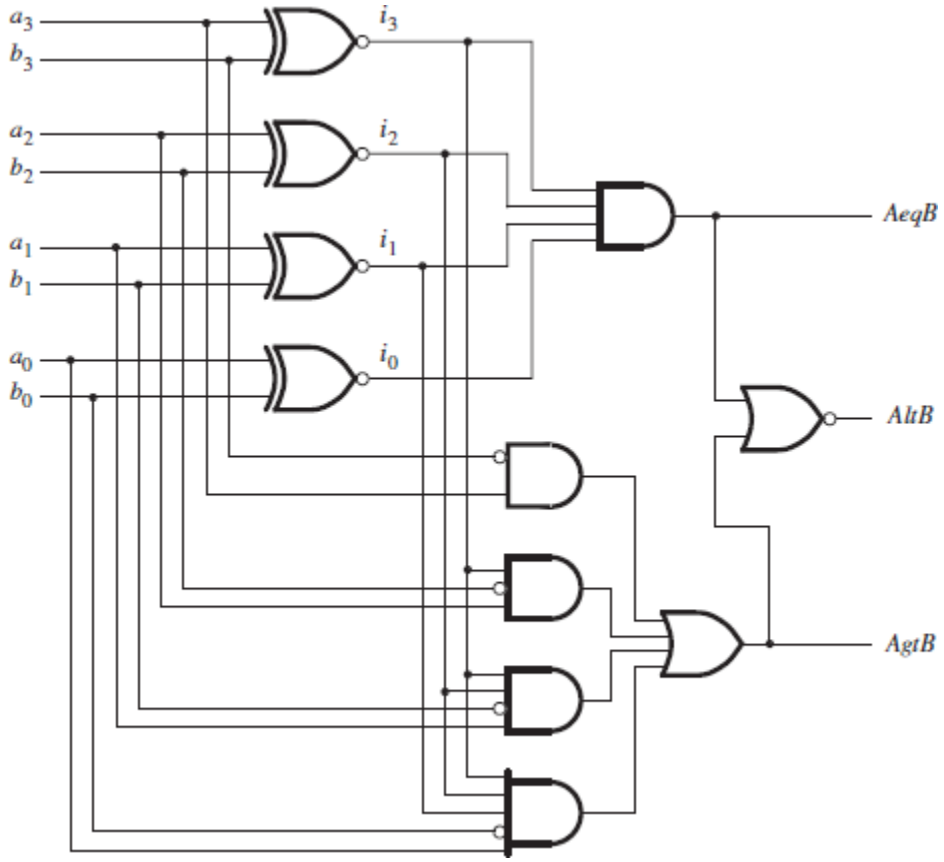


Figure 4.22 A four-bit comparator circuit.

Comparator circuits, like most logic circuits, can be designed in different ways. Another approach for designing a comparator circuit is presented in Example 3.10 in Chapter 3.

4.6 VHDL FOR COMBINATIONAL CIRCUITS

Having presented a number of useful circuits that can be used as building blocks in larger circuits, we will now consider how such circuits can be described in VHDL. Rather than relying on the simple VHDL statements used in previous examples, such as logic expressions, we will specify the

circuits in terms of their behavior. We will also introduce a number of new VHDL constructs.

4.6.1 ASSIGNMENT STATEMENTS

VHDL provides several types of statements that can be used to assign logic values to signals. In the examples of VHDL code given so far, we have mostly relied on simple assignment statements, either for logic or arithmetic expressions. This section discusses other types of assignment statements, which are called selected signal assignments, conditional signal assignments, generate statements, if-then-else statements, and case statements.

4.6.2 SELECTED SIGNAL ASSIGNMENT

A selected signal assignment allows a signal to be assigned one of several values, based on a selection criterion. Figure 4.23 shows how it can be used to describe a 2-to-1 multiplexer. The entity, named *mux2to1*, has the inputs w_0, w_1, and s, and the output f. The selected signal assignment begins with the keyword WITH, which specifies that s is to be used for the selection criterion. The two WHEN clauses state that f is assigned the value of w_0 when $s = 0$; otherwise, f is assigned the value of w_1. The WHEN clause that selects w_1 uses the word OTHERS, instead of the value 1. This is required because the VHDL syntax specifies that a WHEN clause must be included for every possible value of the selection signal s. Since it has the STD_LOGIC type, discussed in Section 2.10.1, s can take the values 0, 1, Z, -, and others. The keyword OTHERS provides a convenient way of accounting for all logic values that are not explicitly listed in a WHEN clause.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
           f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    WITH s SELECT
        f <= w0 WHEN '0',
             w1 WHEN OTHERS ;
END Behavior ;
```

Figure 4.23 VHDL code for a 2-to-1 multiplexer.

Example 4.8

A 4-to-1 multiplexer is described by the entity named *mux4to1* in Figure 4.24. The two select inputs, which are called s_1 and s_0 in Figure 4.2, are represented by the two-bit STD_LOGIC_ VECTOR signal *s*. The selected signal assignment sets *f* to the value of one of the inputs w_0, …, w_3, depending on the valuation of *s*. Compiling the code results in the circuit shown in Figure 4.2*c*. At the end of Figure 4.24, the *mux4to1* entity is defined as a component in the package named *mux4to1_package*. We showed in Section 3.5.2 that declaring the component inside a package makes it convenient to use the entity as a subcircuit in other VHDL code.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT ( w0, w1, w2, w3  : IN    STD_LOGIC ;
           s               : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           f               : OUT   STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Behavior OF mux4to1 IS
BEGIN
    WITH s SELECT
        f <=  w0 WHEN "00",
              w1 WHEN "01",
              w2 WHEN "10",
              w3 WHEN OTHERS ;
END Behavior ;

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
PACKAGE mux4to1_package IS
    COMPONENT mux4to1
        PORT ( w0, w1, w2, w3  : IN    STD_LOGIC ;
               s               : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               f               : OUT   STD_LOGIC ) ;
    END COMPONENT ;
END mux4to1_package ;
```

Figure 4.24 VHDL code for a 4-to-1 multiplexer.

Example 4.9

Figure 4.4 shows how a 16-to-1 multiplexer is built using five 4-to-1 multiplexers. Figure 4.25 presents VHDL code for this circuit, using the *mux4to1* component. The lines of code are numbered so that we can easily refer to them. The *mux4to1_package* is included in the code, because it provides the component declaration for *mux4to1*.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;
3   LIBRARY work ;
4   USE work.mux4to1_package.all ;

5   ENTITY mux16to1 IS
6       PORT ( w  : IN   STD_LOGIC_VECTOR(0 TO 15) ;
7              s  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
8              f  : OUT  STD_LOGIC ) ;
9   END mux16to1 ;

10  ARCHITECTURE Structure OF mux16to1 IS
11      SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
12  BEGIN
13      Mux1: mux4to1 PORT MAP
            ( w(0), w(1), w(2), w(3), s(1 DOWNTO 0), m(0) ) ;
14      Mux2: mux4to1 PORT MAP
            ( w(4), w(5), w(6), w(7), s(1 DOWNTO 0), m(1) ) ;
15      Mux3: mux4to1 PORT MAP
            ( w(8), w(9), w(10), w(11), s(1 DOWNTO 0), m(2) ) ;
16      Mux4: mux4to1 PORT MAP
            ( w(12), w(13), w(14), w(15), s(1 DOWNTO 0), m(3) ) ;
17      Mux5: mux4to1 PORT MAP
            ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
18  END Structure ;
```

Figure 4.25 Hierarchical code for a 16-to-1 multiplexer.

The data inputs to the *mux16to1* entity are the 16-bit signal named *w*, and the select inputs are the four-bit signal named *s*. In the VHDL code signal names are needed for the outputs of the four 4-to-1 multiplexers on the left of Figure 4.4. Line 11 defines a four-bit signal named *m* for this purpose, and lines 13 to 16 instantiate the four multiplexers. For instance, line 13 corresponds to the multiplexer at the top left of Figure 4.4. Its first four ports, which correspond to w_0, …, w_3 in Figure 4.24, are driven by the signals *w*(0), …, *w*(3). The syntax *s*(1 DOWNTO 0) is used to attach the signals *s*(1) and *s*(0) to the two-bit *s* port of the *mux4to1* component. The *m*(0) signal is connected to the multiplexer's output port.

Line 17 instantiates the multiplexer on the right of Figure 4.4. The signals *m*(0), …, *m*(3) are connected to its data inputs, and bits *s*(3) and *s*(2), which are specified by the syntax *s*(3 DOWNTO 2), are attached to the select inputs. The output port generates the *mux16to1* output *f*. Compiling the code results in the multiplexer function

$$f = \overline{s}_3\overline{s}_2\overline{s}_1\overline{s}_0 w_0 + \overline{s}_3\overline{s}_2\overline{s}_1 s_0 w_1 + \overline{s}_3\overline{s}_2 s_1\overline{s}_0 w_2 + \cdots + s_3 s_2 s_1\overline{s}_0 w_{14} + s_3 s_2 s_1 s_0 w_{15}$$

Example 4.10

The selected signal assignment can also be used to describe other types of circuits. Figure 4.26 shows how a selected signal assignment can be used to describe the truth table for a 2-to-4 binary decoder. The entity is called *dec2to4*. The data inputs are the two-bit signal named *w*, and the enable input is *En*. The four outputs are represented by the four-bit signal *y*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN    STD_LOGIC ;
           y   : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
        y <= "1000" WHEN "100",
             "0100" WHEN "101",
             "0010" WHEN "110",
             "0001" WHEN "111",
             "0000" WHEN OTHERS ;
END Behavior ;
```

Figure 4.26 VHDL code for a 2-to-4 binary decoder.

In the truth table for the decoder in Figure 4.14*a*, the inputs are listed in the order *En* w_1 w_0. To represent these three signals, the VHDL code defines the three-bit signal named *Enw*. The statement *Enw* <= *En* & *w* uses the VHDL concatenate operator, which was discussed in Section 3.5.4, to combine the *En* and *w* signals into the *Enw* signal. Hence *Enw*(2)= *En*, *Enw*(1)= w_1, and *Enw*(0) = w_0. The *Enw* signal is used as the selection signal in the selected signal assignment statement. It describes the truth table in Figure 4.14*a*. In the first four WHEN clauses, *En* =1, and the decoder outputs have the same patterns as in the first four rows of the truth

table. The last WHEN clause uses the OTHERS keyword and sets the decoder outputs to 0000, because it represents the cases where *En* =0.

4.6.3 CONDITIONAL SIGNAL ASSIGNMENT

Similar to the selected signal assignment, a conditional signal assignment allows a signal to be set to one of several values. Figure 4.27 shows a modified version of the 2-to-1 multiplexer entity from Figure 4.23. It uses a conditional signal assignment to specify that *f* is assigned the value of w_0 when $s = 0$, or else *f* is assigned the value of w_1. Compiling the code generates the same circuit as the code in Figure 4.23. In this small example the conditional signal assignment has only one WHEN clause. A more complex example, which better illustrates the features of the conditional signal assignment, is given in Example 4.11.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
           f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    f <= w0 WHEN s = '0' ELSE w1 ;
END Behavior ;
```

Figure 4.27 Specification of a 2-to-1 multiplexer using a conditional signal assignment.

Example 4.11

Figure 4.20 gives the truth table for a 4-to-2 priority encoder. VHDL code that describes this truth table is shown in Figure 4.28. The inputs to the encoder are represented by the four-bit signal named *w*. The encoder has the outputs *y*, which is a two-bit signal, and *z*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    y <= "11" WHEN w(3) = '1' ELSE
         "10" WHEN w(2) = '1' ELSE
         "01" WHEN w(1) = '1' ELSE
         "00" ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

Figure 4.28 VHDL code for a priority encoder.

The conditional signal assignment specifies that y is assigned the value 11 when input $w(3) = 1$. If this condition is true, then the other WHEN clauses that follow the ELSE keyword do not affect the value of y. Hence the values of $w(2)$, $w(1)$, and $w(0)$ do not matter, which implements the desired priority scheme. The second WHEN clause states that when $w(2) = 1$, then y is assigned the value 10. This can occur only if $w(3) = 0$. Each successive WHEN clause can affect y only if none of the conditions associated with the preceding WHEN clauses are true. Figure 4.28 includes a second conditional signal assignment for the output z. It states that when all four inputs are 0, z is assigned the value 0; else z is assigned the value 1.

The priority level associated with each WHEN clause in the conditional signal assignment is a key difference from the selected signal assignment, which has no such priority scheme. It is possible to describe the priority encoder using a selected signal assignment, but the code is more awkward. One possibility is shown by the architecture in Figure 4.29. The first WHEN clause sets y to 00 when w_0 is the only input that is 1. The next two clauses state that y should be 01 when $w_3 = w_2 = 0$ and $w_1 = 1$. The next four clauses specify that y should be 10 if $w_3 = 0$ and $w_2 = 1$. Finally, the last WHEN clause states that y should be 1 for all other input valuations, which includes all valuations for which w_3 is 1. Note that the OTHERS clause includes the input valuation 0000. This pattern results in $z = 0$, and the value of y does not matter in this case.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    WITH w SELECT
        y <= "00" WHEN "0001",
             "01" WHEN "0010",
             "01" WHEN "0011",
             "10" WHEN "0100",
             "10" WHEN "0101",
             "10" WHEN "0110",
             "10" WHEN "0111",
             "11" WHEN OTHERS ;
    WITH w SELECT
        z <= '0' WHEN "0000",
             '1' WHEN OTHERS ;
END Behavior ;
```

Figure 4.29 Less efficient code for a priority encoder.

Example 4.12

we derived the circuit for a comparator in Figure 4.22. Figure 4.30 shows how this circuit can be described with VHDL code. Each of the three conditional signal assignments determines the value of one of the comparator outputs. The package named *std_logic_unsigned* is included in the code because it specifies that STD_LOGIC_VECTOR signals, namely, *A* and *B*, can be used as unsigned binary numbers with VHDL relational operators. The relational operators provide a convenient way of specifying the desired functionality.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY compare IS
    PORT ( A, B              : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           AeqB, AgtB, AltB  : OUT  STD_LOGIC ) ;
END compare ;

ARCHITECTURE Behavior OF compare IS
BEGIN
    AeqB <= '1' WHEN A = B ELSE '0' ;
    AgtB <= '1' WHEN A > B ELSE '0' ;
    AltB <= '1' WHEN A < B ELSE '0' ;
END Behavior ;
```

Figure 4.30 VHDL code for a four-bit comparator.

The circuit generated from the code in Figure 4.30 is similar, but not identical, to the circuit in Figure 4.22. The VHDL compiler instantiates a predefined module to implement each of the comparison operations.

Instead of using the *std_logic_unsigned* library, another way to specify that the generated circuit should use unsigned numbers is to include the library named *std_logic_arith.* In this case the signals *A* and *B* should be defined with the type UNSIGNED, rather than STD_LOGIC_VECTOR. If we want the circuit to work with signed numbers, signals *A* and *B* should be defined with the type SIGNED. This code is given in Figure 4.31.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY compare IS
    PORT ( A, B              : IN   SIGNED(3 DOWNTO 0) ;
           AeqB, AgtB, AltB  : OUT  STD_LOGIC ) ;
END compare ;

ARCHITECTURE Behavior OF compare IS
BEGIN
    AeqB <= '1' WHEN A = B ELSE '0' ;
    AgtB <= '1' WHEN A > B ELSE '0' ;
    AltB <= '1' WHEN A < B ELSE '0' ;
END Behavior ;
```

Figure 4.31 The code from Figure 4.30 for signed numbers.

4.6.4 GENERATE STATEMENTS

Figure 4.25 gives VHDL code for a 16-to-1 multiplexer using five instances of a 4-to-1 multiplexer subcircuit. The regular structure of the code suggests that it could be written in a more compact form using a loop. VHDL provides a feature called the FOR GENERATE statement for describing regularly structured hierarchical code.

Figure 4.32 shows the code from Figure 4.25 rewritten using a FOR GENERATE statement. The generate statement must have a label, so we have used the label *G1* in the code. The loop instantiates four copies of the *mux4to1* component, using the loop index *i* in the range from 0 to 3. The variable *i* is not explicitly declared in the code; it is automatically defined as a local variable whose scope is limited to the FOR GENERATE statement. The first loop iteration corresponds to the instantiation statement labeled *Mux1* in Figure 4.25. The * operator represents multiplication; hence for the first loop iteration the VHDL compiler translates the signal names *w*(4*i), *w*(4*i+1), *w*(4*i+2), and *w*(4*i+3) into signal names *w*(0), *w*(1), *w*(2), and *w*(3). The loop iterations for *i*=1, *i*=2, and *i*=3 correspond to the statements labeled *Mux2*, *Mux3*, and *Mux4* in Figure 4.25. The statement labeled *Mux5* in Figure 4.25 does not fit within the loop, so it is included as a separate statement in Figure 4.32. The circuit generated from the code in Figure 4.32 is identical to the circuit produced by using the code in Figure 4.25.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.mux4to1_package.all ;

ENTITY mux16to1 IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(0 TO 15) ;
           s  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           f  : OUT  STD_LOGIC ) ;
END mux16to1 ;

ARCHITECTURE Structure OF mux16to1 IS
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    G1: FOR i IN 0 TO 3 GENERATE
        Muxes: mux4to1 PORT MAP (
            w(4*i), w(4*i+1), w(4*i+2), w(4*i+3), s(1 DOWNTO 0), m(i) ) ;
    END GENERATE ;
    Mux5: mux4to1 PORT MAP ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
END Structure ;
```

Figure 4.32 Code for a 16-to-1 multiplexer using a generate statement.

Example 4.13

In addition to the FOR GENERATE statement, VHDL provides another type of generate statement called IF GENERATE. Figure 4.33 illustrates the use of both types of generate statements. The code shown is a hierarchical description of the 4-to-16 decoder given in Figure 4.16, using five instances of the *dec2to4* component defined in Figure 4.26. The decoder data inputs are the four-bit signal *w*, the enable is *En*, and the outputs are the 16-bit signal *y*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec4to16 IS
    PORT ( w   : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           En  : IN   STD_LOGIC ;
           y   : OUT  STD_LOGIC_VECTOR(0 TO 15) ) ;
END dec4to16 ;

ARCHITECTURE Structure OF dec4to16 IS
    COMPONENT dec2to4
        PORT ( w   : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               En  : IN   STD_LOGIC ;
               y   : OUT  STD_LOGIC_VECTOR(0 TO 3) ) ;
    END COMPONENT ;
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    G1: FOR i IN 0 TO 3 GENERATE
        Dec_ri: dec2to4 PORT MAP ( w(1 DOWNTO 0), m(i), y(4*i TO 4*i+3));
        G2: IF i=3 GENERATE
            Dec_left: dec2to4 PORT MAP ( w(i DOWNTO i-1), En, m ) ;
        END GENERATE ;
    END GENERATE ;
END Structure ;
```

Figure 4.33 Hierarchical code for a 4-to-16 binary decoder.

Following the component declaration for the *dec2to4* subcircuit, the architecture defines the signal *m*, which represents the outputs of the 2-to-4 decoder on the left of Figure 4.16. The five copies of the *dec2to4* component are instantiated by the FOR GENERATE statement. In each iteration of the loop, the statement labeled *Dec_ri* instantiates a *dec2to4* component that corresponds to one of the 2-to-4 decoders on the right side

of Figure 4.16. The first loop iteration generates the *dec2to4* component with data inputs w_1 and w_0, enable input m_0, and outputs y_0, y_1, y_2, y_3. The other loop iterations also use data inputs w_1 w_0, but use different bits of m and y.

The IF GENERATE statement, labeled *G2*, instantiates a *dec2to4* component in the last loop iteration, for which the condition i=3 is true. This component represents the 2-to-4 decoder on the left of Figure 4.16. It has the two-bit data inputs w_3 and w_2, the enable *En*, and the outputs m_0, m_1, m_2, and m_3. Note that instead of using the IF GENERATE statement, we could have instantiated this component outside the FOR GENERATE statement. We have written the code as shown simply to give an example of the IF GENERATE statement.

The generate statements in Figures 4.32 and 4.33 are used to instantiate components. Another use of generate statements is to generate a set of logic equations. An example of this use will be given in Figure 7.13.

4.6.5 Concurrent and Sequential Assignment Statements

We have introduced several types of assignment statements: simple assignment statements, which involve logic or arithmetic expressions, selected assignment statements, and conditional assignment statements. All of these statements share the property that the order in which they appear in VHDL code does not affect the meaning of the code. Because of this property, these statements are called the *concurrent assignment statements*.

VHDL also provides a second category of statements, called *sequential assignment statements*, for which the ordering of the statements may affect the meaning of the code. We will discuss two types of sequential assignment statements: if-then-else statements, introduced in Section 2.10.2, and case statements. As mentioned in Section 2.10.2, VHDL requires that the sequential assignment statements be placed inside another type of statement, called a process statement.

4.6.6 Process Statement

Figures 4.23 and 4.27 show two ways of describing a 2-to-1 multiplexer, using the selected and conditional signal assignments. The same circuit can

also be described using the if-then-else statement in Figure 4.34. The process statement, or simply *process*, that begins with the PROCESS keyword is followed by the parenthesized list of signals in the *sensitivity list*. For a combinational circuit like the multiplexer, the sensitivity list includes all input signals that are used inside the process. The process statement is translated by the VHDL compiler into logic expressions. In the figure the process consists of the single if-then-else statement that describes the multiplexer function. Thus the sensitivity list comprises the data inputs, w_0 and w_1, and the select input s.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN   STD_LOGIC ;
           f          : OUT  STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        IF s = '0' THEN
            f <= w0 ;
        ELSE
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 4.34 A 2-to-1 multiplexer specified using the if-then-else statement.

In general, there can be a number of statements inside a process. These statements are considered as follows. Using VHDL jargon, we say that when there is a change in the value of any signal in the process's sensitivity list, then the process becomes *active*. Once active, the statements inside the process are evaluated in sequential order. Any assignments made to signals inside the process are not visible outside the process until all of the statements in the process have been evaluated. If there are multiple assignments to the same signal, only the last one has any visible effect. This is illustrated in Example 4.14.

Example 4.14

The code in Figure 4.35 is equivalent to the code in Figure 4.34. The first statement in the process assigns the value of w_0 to f. This provides a *default* value for f but the assignment does not actually take place until the end of the process. In VHDL jargon we say that the assignment is *scheduled* to occur after all of the statements in the process have been evaluated. If another assignment to f takes place while the process is active, the default assignment will be overridden. The second statement in the process assigns the value of w_1 to f if the value of s is equal to 1. If this condition is true, then the default assignment is overridden. Thus if s=0, then f=w_0, and if s=1, then f=w_1, which defines the 2-to-1 multiplexer. Compiling this code results in the same circuit as for Figures 4.23, 4.27, and 4.34, namely, $f = \bar{s}w_0 + sw_1$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN   STD_LOGIC ;
           f          : OUT  STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        f <= w0 ;
        IF s = '1' THEN
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 4.35 Alternative code for the 2-to-1 multiplexer using an if statement.

The process statement in Figure 4.35 illustrates that the ordering of the statements in a process can affect the meaning of the code. Consider reversing the order of the two statements so that the if-then-else statement is evaluated first. If s=1, f is assigned the value of w_1. This assignment is scheduled and does not take place until the end of the process. However, the statement f <= w_0 is evaluated last. It overrides the first assignment, and f is

assigned the value of w_0 regardless of the value of s. Hence instead of describing a multiplexer, when the statements inside the process are reversed, the code represents the trivial circuit $f=w_0$.

Example 4.15

Figure 4.36 gives an example that contains both a concurrent assignment statement and a process statement. It describes a priority encoder and is equivalent to the code in Figure 4.28. The process describes the desired priority scheme using an if-then-else statement. It specifies that if the input w_3 is 1, then the output is set to y = 11. This assignment does not depend on the values of inputs w_2, w_1, or w_0; hence their values do not matter. The other clauses in the if-then-else statement are evaluated only if w_3 = 0. The first ELSIF clause states that if w_2 is 1, then y = 10. If w_2 = 0, then the next ELSIF clause results in y = 01 if w_1 = 1. If w_3=w_2=w_1=0, then the ELSE clause results in y=00. This assignment is done whether or not w_0 is 1; Figure 4.24 indicates that y can be set to any pattern when w=0000 because z will be set to 0 in this case.

The priority encoder's output z must be set to 1 whenever at least one of the data inputs is 1. This output is defined by the conditional assignment statement at the end of Figure 4.36. The VHDL syntax does not allow a conditional assignment statement (or a selected assignment statement) to appear inside a process. An alternative would be to specify the value of z by using an if-then-else statement inside the process. The reason that we have written the code as given in the figure is to illustrate that concurrent assignment statements can be used in conjunction with process statements. The process statement serves the purpose of separating the sequential statements from the concurrent statements. Note that the ordering of the process statement and the conditional assignment statement does not matter. VHDL stipulates that while the statements inside a process are sequential statements, the process statement itself is a concurrent statement.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS ( w )
    BEGIN
        IF w(3) = '1' THEN
            y <= "11" ;
        ELSIF w(2) = '1' THEN
            y <= "10" ;
        ELSIF w(1) = '1' THEN
            y <= "01" ;
        ELSE
            y <= "00" ;
        END IF ;
    END PROCESS ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

Figure 4.36 A priority encoder specified using the if-then-else statement.

Example 4.16

Figure 4.37 shows an alternative style of code for the priority encoder, using if-then-else statements. The first statement in the process provides the default value of 00 for $y_1\ y_0$. The second statement overrides this if w_1 is 1, and sets $y_1\ y_0$ to 01. Similarly, the third and fourth statements override the previous ones if w_2 or w_3 are 1, and set $y_1\ y_0$ to 10 and 11, respectively. These four statements are equivalent to the single if-then-else statement in Figure 4.36 that describes the priority scheme. The value of z is specified using a default assignment statement, followed by an if-then-else statement that overrides the default if w=0000. Although the examples in Figures 4.36 and 4.37 are equivalent, the meaning of the code in Figure 4.36 is probably easier to understand.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS ( w )
    BEGIN
        y <= "00" ;
        IF w(1) = '1' THEN y <= "01" ; END IF ;
        IF w(2) = '1' THEN y <= "10" ; END IF ;
        IF w(3) = '1' THEN y <= "11" ; END IF ;

        z <= '1' ;
        IF w = "0000" THEN z <= '0' ; END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 4.37 Alternative code for the priority encoder.

Example 4.17

Figure 4.30 specifies a four-bit comparator that produces the three outputs *AeqB*, *AgtB*, and *AltB*. Figure 4.38 shows how such specification can be written using if-then-else statements. For simplicity, one-bit numbers are used for the inputs *A* and *B*, and only the code for the *AeqB* output is shown. The process assigns the default value of 0 to *AeqB* and then the if-then-else statement changes *AeqB* to 1 if *A* and *B* are equal. It is instructive to consider the effect on the semantics of the code if the default assignment statement is removed, as illustrated in Figure 4.39.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY compare1 IS
    PORT ( A, B   : IN     STD_LOGIC ;
           AeqB : OUT  STD_LOGIC ) ;
END compare1 ;

ARCHITECTURE Behavior OF compare1 IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        AeqB <= '0' ;
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 4.38 Code for a one-bit equality comparator.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
    PORT ( A, B   : IN     STD_LOGIC ;
           AeqB : OUT  STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 4.39 An example of code that results in implied memory.

With only the if-then-else statement, the code does not specify what value *AeqB* should have if the condition $A = B$ is not true. The VHDL semantics stipulate that in cases where the code does not specify the value of a signal, the signal should retain its current value. For the code in Figure 4.39, once *A* and *B* are equal, resulting in *AeqB* =1, then *AeqB* will remain set to 1 indefinitely, even if *A* and *B* are no longer equal. In the VHDL jargon, the *AeqB* output is said to have *implied memory* because the circuit

synthesized from the code will "remember," or store the value *AeqB* =1. Figure 4.40 shows the circuit synthesized from the code. The XOR gate produces a 1 when *A* and *B* are equal, and the OR gate ensures that *AeqB* remains set to 1 indefinitely.

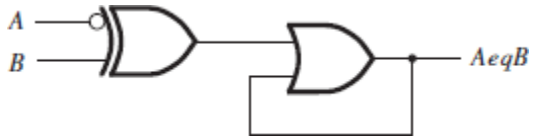


Figure 4.40 The circuit generated from the code in Figure 4.39.

The implied memory that results from the code in Figure 4.39 is not useful, because it generates a comparator circuit that does not function correctly. However, we will show in Chapter 5 that the semantics of implied memory are useful for other types of circuits, which have the capability to store logic signal values in memory elements.

4.6.7 CASE STATEMENT

A case statement is similar to a selected signal assignment in that the case statement has a selection signal and includes WHEN clauses for various valuations of this selection signal. Figure 4.41 shows how the case statement can be used as yet another way of describing the 2-to-1 multiplexer circuit. The case statement begins with the CASE keyword, which specifies that *s* is to be used as the selection signal. The first WHEN clause specifies, following the => designator, the statements that should be evaluated when *s*=0. In this example the only statement evaluated when *s*=0 is $f <= w_0$. The case statement must include a WHEN clause for all possible valuations of the selection signal. Hence the second WHEN clause, which contains $f <= w_1$, uses the OTHERS keyword.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN     STD_LOGIC ;
           f          : OUT    STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        CASE s IS
            WHEN '0' =>
                f <= w0 ;
            WHEN OTHERS =>
                f <= w1 ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 4.41 A case statement that represents a 2-to-1 multiplexer.

Example 4.18

Figure 4.26 gives the code for a 2-to-4 decoder. A different way of describing this circuit, using sequential assignment statements, is shown in Figure 4.42. The process first uses an if-then-else statement to check the value of the decoder enable signal *En*. If *En* = 1, the case statement sets the output *y* to the appropriate value based on the input *w*. The case statement represents the first four rows of the truth table in Figure 4.14*a*. If *En* = 0, the ELSE clause sets *y* to 0000, as specified in the bottom row of the truth table.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN    STD_LOGIC ;
           y   : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
BEGIN
    PROCESS ( w, En )
    BEGIN
        IF En = '1' THEN
            CASE w IS
                WHEN "00" =>
                    y <= "1000" ;
                WHEN "01" =>
                    y <= "0100" ;
                WHEN "10" =>
                    y <= "0010" ;
                WHEN OTHERS =>
                    y <= "0001" ;
            END CASE ;
        ELSE
            y <= "0000" ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 4.42 A process statement that describes a 2-to-4 binary decoder.

Example 4.19

Another example of a case statement is given in Figure 4.43. The entity is named *seg7*, and it represents the hex-to-7-segment decoder in Figure 4.21. The hexadecimal input is represented by the four-bit signal named *hex*, and the seven outputs are the seven-bit signal named *leds*. The case statement is formatted so that it resembles the truth table in Figure 4.21*c*. Note that there is a comment to the right of the case statement, which labels the seven outputs with the letters from *a* to *g*. These labels indicate the correlation between the seven-bit *leds* signal in the VHDL code and the seven segments in Figure 4.21*b*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY seg7 IS
    PORT ( hex   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           leds  : OUT   STD_LOGIC_VECTOR(1 TO 7) ) ;
END seg7 ;

ARCHITECTURE Behavior OF seg7 IS
BEGIN
    PROCESS ( hex )
    BEGIN
        CASE hex IS                    --   abcdefg
            WHEN "0000"    => leds <= "1111110" ;
            WHEN "0001"    => leds <= "0110000" ;
            WHEN "0010"    => leds <= "1101101" ;
            WHEN "0011"    => leds <= "1111001" ;
            WHEN "0100"    => leds <= "0110011" ;
            WHEN "0101"    => leds <= "1011011" ;
            WHEN "0110"    => leds <= "1011111" ;
            WHEN "0111"    => leds <= "1110000" ;
            WHEN "1000"    => leds <= "1111111" ;
            WHEN "1001"    => leds <= "1111011" ;
            WHEN "1010"    => leds <= "1110111" ;
            WHEN "1011"    => leds <= "0011111" ;
            WHEN "1100"    => leds <= "1001110" ;
            WHEN "1101"    => leds <= "0111101" ;
            WHEN "1110"    => leds <= "1001111" ;
            WHEN OTHERS => leds <= "1000111" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 4.43 Code that represents a hex-to-7-segment decoder.

Example 4.20

An arithmetic logic unit (ALU) is a logic circuit that performs various Boolean and arithmetic operations on *n*-bit operands. Table 4.1 specifies the functionality of a simple ALU, known as the 74381 chip, which has been available in the form of a standard chip in the family called the 7400-series. Such chips are discussed in Appendix B. This ALU has 2 four-bit data inputs, *A* and *B*, a three-bit select input, *S*, and a four-bit output, *F*. As the table shows, *F* is defined by various arithmetic or Boolean operations on the inputs *A* and *B*. In this table + means arithmetic addition, and - means

arithmetic subtraction. To avoid confusion, the table uses the words XOR, OR, and AND for the Boolean operations. Each Boolean operation is done in a bit-wise fashion. For example, $F = A$ AND B produces the four-bit result $f_0 = a_0\ b_0$, $f_1 = a_1\ b_1$, $f_2 = a_2 b_2$, and $f_3 = a_3 b_3$.

Table 4.1 The functionality of the 74381 ALU.

Operation	Inputs $s_2\ s_1\ s_0$	Outputs F
Clear	0 0 0	0 0 0 0
B–A	0 0 1	$B - A$
A–B	0 1 0	$A - B$
ADD	0 1 1	$A + B$
XOR	1 0 0	A XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

Figure 4.44 shows how the functionality of the 74381 ALU can be described using VHDL code. The std_logic_unsigned package, introduced in Section 3.5.4, is included so that the STD_LOGIC_VECTOR signals A and B can be used in unsigned arithmetic operations. The case statement shown corresponds directly to Table 4.1.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY alu IS
    PORT ( s      : IN    STD_LOGIC_VECTOR(2 DOWNTO 0) ;
           A, B   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           F      : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END alu ;

ARCHITECTURE Behavior OF alu IS
BEGIN
    PROCESS ( s, A, B )
    BEGIN
        CASE s IS
            WHEN "000" =>
                F <= "0000" ;
            WHEN "001" =>
                F <= B - A ;
            WHEN "010" =>
                F <= A - B ;
            WHEN "011" =>
                F <= A + B ;
            WHEN "100" =>
                F <= A XOR B ;
            WHEN "101" =>
                F <= A OR B ;
            WHEN "110" =>
                F <= A AND B ;
            WHEN OTHERS =>
                F <= "1111" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 4.44 Code that represents the functionality of the 74381 ALU chip.

4.6.8 VHDL OPERATORS

In this section we discuss the VHDL operators that are useful for synthesizing logic circuits. Table 4.2 lists these operators in groups that reflect the type of operation performed.

Table 4.2 VHDL operators (used for synthesis).

Operator category	Operator symbol	Operation performed
Logical	AND	AND
	OR	OR
	NAND	Not AND
	NOR	Not OR
	XOR	XOR
	XNOR	Not XOR
	NOT	NOT
Relational	=	Equality
	/=	Inequality
	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
Arithmetic	+	Addition
	–	Subtraction
	*	Multiplication
	/	Division
Concatenation	&	Concatenation
Shift and Rotate	SLL	Shift left logical
	SRL	Shift right logical
	SLA	Shift left arithmetic
	SRA	Shift right arithmetic
	ROL	Rotate left
	ROR	Rotate right

To illustrate the results produced by the various operators, we will use three-bit vectors A(2 DOWNTO 0), B(2 DOWNTO 0), and C(2 DOWNTO 0).

Logical Operators

The logical operators can be used with bit and boolean types of operands. The operands can be either single-bit scalars or multibit vectors. For example, the statement

```
C <= NOT A;
```

produces the result $c_2 = \overline{a}_2, c_1 = \overline{a}_1$, and $c_0 = \overline{a}_0$, where a_i and c_i are the bits of the vectors *A* and *C*.

The statement

```
C <= A AND B;
```

generates $c_2 = a_2 \cdot b_2$, $c_1 = a_1 \cdot b_1$, and $c_0 = a_0 \cdot b_0$. The other operators lead to similar evaluations.

Relational Operators

The relational operators are used to compare expressions. The result of the comparison is TRUE or FALSE. The expressions that are compared must be of the same type. For example, if *A* = 011 and *B* = 010 then A > B evaluates to TRUE, and B/ = “010” evaluates to FALSE.

Arithmetic Operators

We have already encountered the arithmetic operators in Chapter 3. They performstandard arithmetic operations. Thus

```
C <= A + B;
```

puts the three-bit sum of A plus B into C, while

```
C <= A – B;
```

puts the difference of *A* and *B* into *C*. The operation

```
C <= –A;
```

places the 2's complement of *A* into *C*.

The addition, subtraction, and multiplication operations are supported by most CAD synthesis tools. However, the division operation is often not supported. When the VHDL compiler encounters an arithmetic operator, it usually synthesizes it by using an appropriate module from a library.

Concatenate Operator

This operator concatenates two or more vectors to create a larger vector. For example,

```
D <= A & B;
```

defines the six-bit vector $D = a_2a_1a_0b_2b_1b_0$. Similarly, the concatenation

```
E <= "111" & A & "00" ;
```

produces the eight-bit vector $E = 111a_2a_1a_000$.

Shift and Rotate Operators

A vector operand can be shifted to the right or left by a number of bits specified as a constant. When bits are shifted, the vacant bit positions are filled with 0s. For example,

```
B <= A SLL 1;
```

results in $b_2 = a_1$, $b_1 = a_0$, and $b_0 = 0$. Similarly,

```
B <= A SRL 2;
```

yields $b_2 = b_1 = 0$ and $b_0 = a_2$.

The arithmetic shift left, SLA, has the same effect as SLL. But, the arithmetic shift right, SRA, performs the sign extension by replicating the sign bit into the positions left vacant after shifting. Hence

```
B <= A SRA 1;
```

gives $b_2 = a_2$, $b_1 = a_2$, and $b_0 = a_1$.

An operand can also be rotated, in which case the bits shifted out from one end are placed into the vacated positions at the other end. For example,

```
B <= A ROR 2;
```

produces $b_2 = a_1$, $b_1 = a_0$, and $b_0 = a_2$.

Operator Precedence

Operators in different categories have different precedence. Operators in the same category have the same precedence, and are evaluated from left to right in a given expression. It is a good practice to use parentheses to indicate the desired order of operations in the expression. To illustrate this point, consider the statement

```
S <= A + B + C + D;
```

which defines the addition of four vector operands. The VHDL compiler will synthesize a circuit as if the expression was written in the form (*A* + *B*) + *C*) + *D*, which gives a cascade of three adders so that the final sum will be available after a propagation delay through three adders. By writing the statement as

```
S <= (A + B) + (C + D);
```

the synthesized circuit will still have three adders, but since the sums *A* + *B* and *C* + *D* are generated in parallel, the final sum will be available after a propagation delay through only two adders.

Table 4.2 groups the operators informally according to their functionality. It shows only those operators that are used to synthesize logic circuits. The VHDL Standard specifies additional operators, which are useful for simulation and documentation purposes. All operators are grouped into different classes, with a defined precedence ordering between classes. We discuss this issue in Appendix A, Section A.3.

4.7 Concluding Remarks

This chapter has introduced a number of circuit building blocks. Examples using these blocks to construct larger circuits will be presented in later chapters. To describe the building block circuits efficiently, several VHDL constructs have been introduced. In many cases a given circuit can be described in various ways, using different constructs. A circuit that can be described using a selected signal assignment can also be described using a case statement. Circuits that fit well with conditional signal assignments are also well-suited to if-then-else statements. In general, there are no clear rules that dictate when one type of assignment statement should be preferred over another. With experience the user develops a sense for which types of statements work well in a particular design situation. Personal preference also influences how the code is written.

VHDL is not a programming language, and VHDL code should not be written as if it were a computer program. The concurrent and sequential

assignment statements discussed in this chapter can be used to create large, complex circuits. A good way to design such circuits is to construct them using well-defined modules, in the manner that we illustrated for the multiplexers, decoders, encoders, and so on. Additional examples using the VHDL statements introduced in this chapter are given in Chapters 5 and 6. In Chapter 7 we provide a number of examples of using VHDL code to describe larger digital systems. For more information on VHDL, the reader can consult more specialized books.

4.8 Examples of Solved Problems

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

Example 4.21

Problem: Implement the function $f(w_1, w_2, w_3) = \sum m(0, 1, 3, 4, 6, 7)$ by using a 3-to-8 binary decoder and an OR gate.

Solution: The decoder generates a separate output for each minterm of the required function. These outputs are then combined in the OR gate, giving the circuit in Figure 4.45.

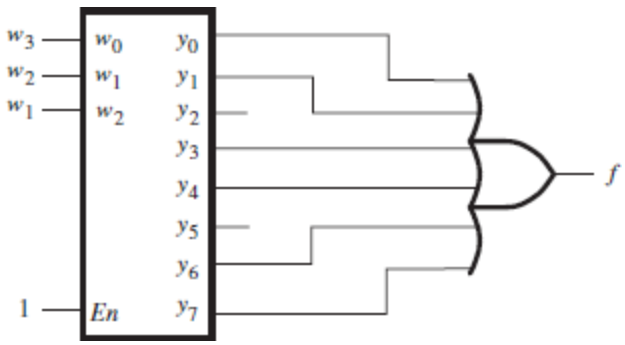


Figure 4.45 Circuit for Example 4.21.

Example 4.22

Problem: Derive a circuit that implements an 8-to-3 binary encoder.

Solution: The truth table for the encoder is shown in Figure 4.46. Only those rows for which a single input variable is equal to 1 are shown; the other rows can be treated as don't care cases. From the truth table it is seen that the desired circuit is defined by the equations

$$y_2 = w_4 + w_5 + w_6 + w_7$$
$$y_1 = w_2 + w_3 + w_6 + w_7$$
$$y_0 = w_1 + w_3 + w_5 + w_7$$

w_7	w_6	w_5	w_4	w_3	w_2	w_1	w_0	y_2	y_1	y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Figure 4.46 Truth table for an 8-to-3 binary encoder.

Example 4.23

Problem: Implement the function

$$f(w_1, w_2, w_3, w_4, w_5) = \overline{w}_1\overline{w}_2\overline{w}_4\overline{w}_5 + w_1w_2 + w_1w_3 + w_1w_4 + w_3w_4w_5$$

by using a 4-to-1 multiplexer and as few other gates as possible. Assume that only the uncomplemented inputs w_1, w_2, w_3, w_4, and w_5 are available.

Solution: Since variables w_1 and w_4 appear in more product terms in the expression for f than the other three variables, let us perform Shannon's expansion with respect to these two variables. The expansion gives

$$\begin{aligned} f &= \overline{w}_1\overline{w}_4 f_{\overline{w}_1\overline{w}_4} + \overline{w}_1 w_4 f_{\overline{w}_1 w_4} + w_1\overline{w}_4 f_{w_1\overline{w}_4} + w_1 w_4 f_{w_1 w_4} \\ &= \overline{w}_1\overline{w}_4(\overline{w}_2\overline{w}_5) + \overline{w}_1 w_4(w_3 w_5) + w_1\overline{w}_4(w_2 + w_3) + w_1 w_4(1) \end{aligned}$$

We can use a NOR gate to implement $\overline{w}_2\overline{w}_5 = \overline{w_2 + w_5}$. We also need an AND gate and an OR gate. The complete circuit is presented in Figure 4.47.

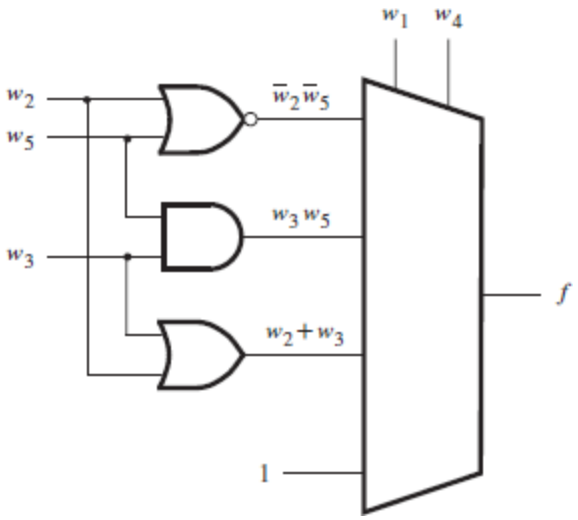


Figure 4.47 Circuit for Example 4.23.

Example 4.24

Problem: In Chapter 2 we pointed out that the rows and columns of a Karnaugh map are labeled using Gray code. This is a code in which consecutive valuations differ in one variable only. Figure 4.48 depicts the conversion between three-bit binary and Gray codes. Design a circuit that can convert a binary code into Gray code according to the figure.

b_2	b_1	b_0	g_2	g_1	g_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

Figure 4.48 Binary to Gray code conversion.

Solution: From the figure it follows that

$$
\begin{aligned}
g_2 &= b_2 \\
g_1 &= b_1\overline{b}_2 + \overline{b}_1 b_2 \\
&= b_1 \oplus b_2 \\
g_0 &= b_0\overline{b}_1 + \overline{b}_0 b_1 \\
&= b_0 \oplus b_1
\end{aligned}
$$

Example 4.25

Problem: In Section 4.1.2 we showed that any logic function can be decomposed using Shannon's expansion theorem. For a four-variable function, $f(w_1, \ldots, w_4)$, the expansion with respect to w_1 is

$$f(w_1, \ldots, w_4) = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

A circuit that implements this expression is given in Figure 4.49*a*.

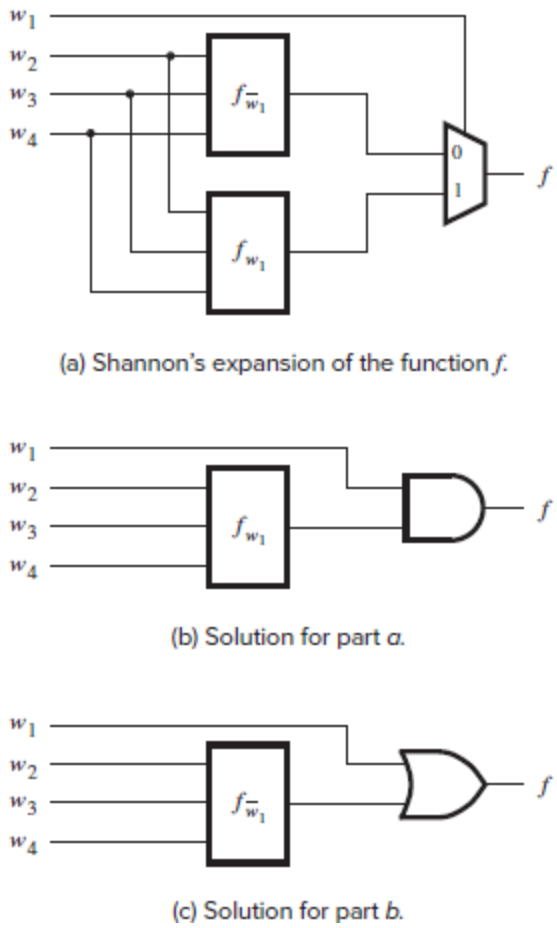


Figure 4.49 Circuits for Example 4.25.

(a) If the decomposition yields $f_{\bar{w}_1} = 0$, then the multiplexer in the figure can be replaced by a single logic gate. Show this circuit.

(b) Repeat part (*a*) for the case where $f_{w_1} = 1$.

Solution: The desired circuits are shown in parts (*b*) and (*c*) of Figure 4.49.

Example 4.26

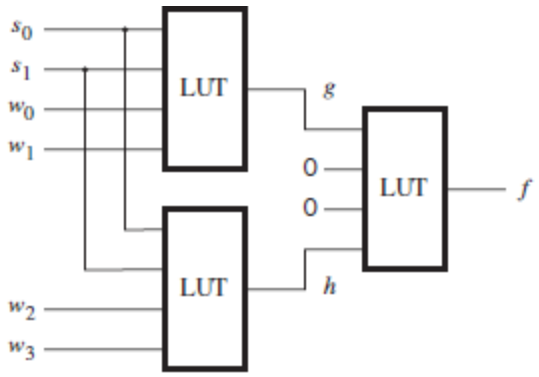
Problem: In Section 2.17 we said that field-programmable gate arrays (FPGAs) contain lookup tables (LUTs) that are used to implement logic functions. Each LUT can be programmed to implement any logic function of its inputs. FPGAs are discussed in detail in Appendix B. Many commercial FPGAs contain four-input lookup tables (4-LUTs). What is the

minimum number of 4-LUTs needed to construct a 4-to-1 multiplexer with select inputs s_1 and s_0 and data inputs w_3, w_2, w_1, and w_0?

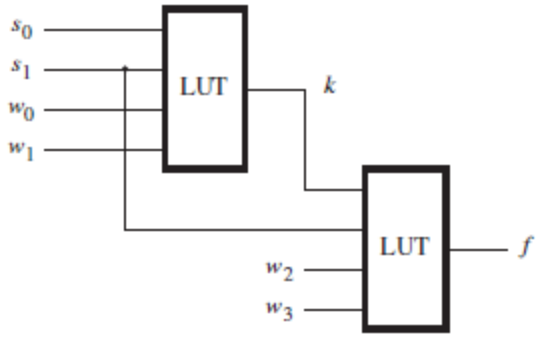
Solution: A straightforward attempt is to use directly the expression that defines the 4-to-1 multiplexer

$$f = \bar{s}_1\bar{s}_0w_0 + \bar{s}_1s_0w_1 + s_1\bar{s}_0w_2 + s_1s_0w_3$$

Let $g = \bar{s}_1\bar{s}_0w_0 + \bar{s}_1s_0w_1$ and $h = s_1\bar{s}_0w_2 + s_1s_0w_3$, so that $f = g + h$. This decomposition leads to the circuit in Figure 4.50*a*, which requires three LUTs.



(a) Using three LUTs



(b) Using two LUTs

Figure 4.50 Circuits for Example 4.26.

When designing logic circuits, one can sometimes come up with a clever idea which leads to a superior implementation. Figure 4.50*b* shows how it is possible to implement the multiplexer with just two LUTs, based on the following observation. The truth table in Figure 4.2*b* indicates that when $s_1 = 0$ the output must be either w_0 or w_1, as determined by the value of s_0.

This can be generated by the first LUT, with the output k. The second LUT must make the choice between w_2 and w_3 when $s_1 = 1$. But, the choice can be made only by knowing the value of s_0. Since it is impossible to have five inputs in the LUT, more information has to be passed from the first to the second LUT. Observe that when $s_1 = 1$ the output f will be equal to either w_2 or w_3, in which case it is not necessary to know the values of w_0 and w_1. Hence, in this case we can pass on the value of s_0 through the first LUT, rather than w_0 or w_1. This can be done by making the function of this LUT

$$k = \overline{s}_1(\overline{s}_0 w_0 + s_0 w_1) + s_1 s_0$$

Then, the second LUT performs the function

$$f = \overline{s}_1 k + s_1(\overline{k} w_2 + k w_3)$$

Example 4.27

Problem: In digital systems it is often necessary to have circuits that can shift the bits of a vector by one or more bit positions to the left or right. Design a circuit that can shift a four-bit vector $W = w_3 w_2 w_1 w_0$ one bit position to the right when a control signal *Shift* is equal to 1. Let the outputs of the circuit be a four-bit vector $Y = y_3\, y_2\, y_1\, y_0$ and a signal k, such that if *Shift* = 1 then $y_3 = 0$, $y_2 = w_3$, $y_1 = w_2$, $y_0 = w_1$, and $k = w_0$. If *Shift* = 0 then $Y = W$ and $k = 0$.

Solution: The required circuit can be implemented with five 2-to-1 multiplexers as shown in Figure 4.51. The *Shift* signal is used as the select input to each multiplexer.

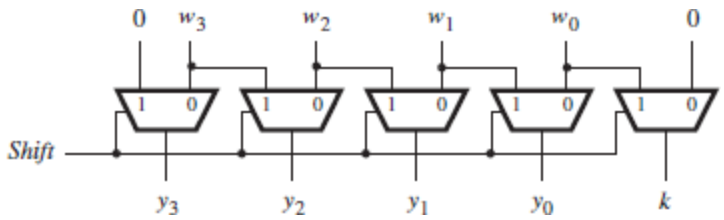


Figure 4.51 A shifter circuit.

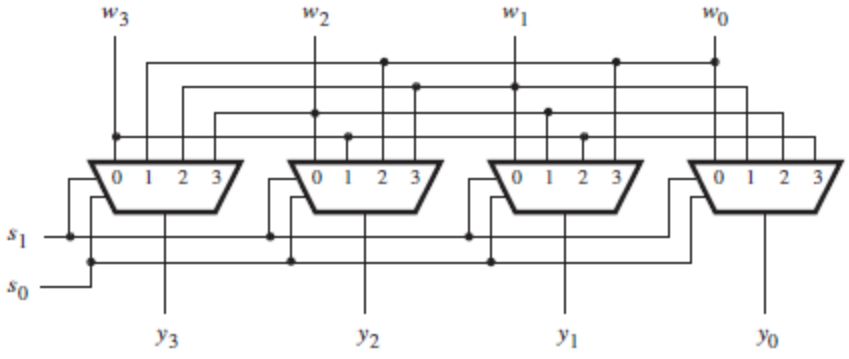
Example 4.28

Problem: The shifter circuit in Example 4.30 shifts the bits of an input vector by one bit position to the right. It fills the vacated bit on the left side with 0. A more versatile shifter circuit may be able to shift by more bit positions at a time. If the bits that are shifted out are placed into the vacated positions on the left, then the circuit effectively rotates the bits of the input vector by a specified number of bit positions. Such a circuit is often called a *barrel shifter*. Design a four-bit barrel shifter that rotates the bits by 0, 1, 2, or 3 bit positions as determined by the valuation of two control signals s_1 and s_0.

Solution: The required action is given in Figure 4.52*a*. The barrel shifter can be implemented with four 4-to-1 multiplexers as shown in Figure 4.52*b*. The control signals s_1 and s_0 are used as the select inputs to the multiplexers.

s_1	s_0	y_3	y_2	y_1	y_0
0	0	w_3	w_2	w_1	w_0
0	1	w_0	w_3	w_2	w_1
1	0	w_1	w_0	w_3	w_2
1	1	w_2	w_1	w_0	w_3

(a) Truth table



(b) Circuit

Figure 4.52 A barrel shifter circuit.

Example 4.29

Problem: Write VHDL code that represents the circuit in Figure 4.17. Use the *dec2to4* entity in Figure 4.26 as a subcircuit in your code.

Solution: The code is shown in Figure 4.53. Note that the *dec2to4* entity can be included in the same file as we have done in the figure, but it can also be in a separate file in the project directory.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT ( s   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           w   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           f   : OUT   STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Structure OF mux4to1 IS
    COMPONENT dec2to4
        PORT ( w  : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               En : IN    STD_LOGIC ;
               y  : OUT   STD_LOGIC_VECTOR(0 TO 3));
    END COMPONENT;
    SIGNAL y : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    decoder: dec2to4 PORT MAP (s, '1' , y) ;
    f <= (w(0) AND y(0)) OR (w(1) AND y(1)) OR (w(2) AND y(2)) OR (w(3) AND y(3)) ;
END Structure ;


LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN    STD_LOGIC ;
           y   : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
        y <=  "1000" WHEN "100" ,
              "0100" WHEN "101" ,
              "0010" WHEN "110" ,
              "0001" WHEN "111" ,
              "0000" WHEN OTHERS ;
END Behavior ;
```

Figure 4.53 VHDL code for Example 4.29.

Example 4.30

Problem: Write VHDL code that represents the shifter circuit in Figure 4.51.

Solution: There are two possible approaches: structural and behavioral. A structural description is given in Figure 4.54. The IF construct is used to define the desired shifting of individual bits. A typical VHDL compiler will implement this code with 2-to-1 multiplexers as depicted in Figure 4.51.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shifter IS
    PORT ( w       : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Shift   : IN    STD_LOGIC ;
           y       : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           k       : OUT   STD_LOGIC ) ;
END shifter ;

ARCHITECTURE Behavior OF shifter IS
BEGIN
    PROCESS (Shift, w)
    BEGIN
        IF Shift = '1' THEN
            y(3) <= '0' ;
            y(2 DOWNTO 0) <= w(3 DOWNTO 1) ;
            k <= w(0) ;
        ELSE
            y <= w ;
            k <= '0' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 4.54 Structural VHDL code for the circuit in Figure 4.51.

A behavioral specification is given in Figure 4.55. It makes use of the shift operator SRL. Since the shift and rotate operators are supported in the *ieee.numeric_std.all* library, this library must be included in the code. Note that the vectors *w* and *y* are defined to be of UNSIGNED type.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

USE ieee.numeric_std.all ;

ENTITY shifter IS
    PORT ( w     : IN   UNSIGNED(3 DOWNTO 0) ;
           Shift : IN   STD_LOGIC ;
           y     : OUT  UNSIGNED(3 DOWNTO 0) ;
           k     : OUT  STD_LOGIC ) ;
END shifter ;

ARCHITECTURE Behavior OF shifter IS
BEGIN
    PROCESS ( Shift, w )
    BEGIN
        IF Shift = '1' THEN
            y <= w SRL 1 ;
            k <= w(0) ;
        ELSE
            y <= w ;
            k <= '0' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 4.55 Behavioral VHDL code for the circuit in Figure 4.51.

Example 4.31

Problem: Write VHDL code that defines the barrel shifter in Figure 4.52.

Solution: An easy way to specify the barrel shifter is by using the VHDL rotate operator. The complete code is presented in Figure 4.56.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

USE ieee.numeric_std.all ;

ENTITY barrel IS
    PORT ( w  : IN   UNSIGNED(3 DOWNTO 0) ;
           s  : IN   UNSIGNED(1 DOWNTO 0) ;
           y  : OUT  UNSIGNED(3 DOWNTO 0) ) ;
END barrel ;

ARCHITECTURE Behavior OF barrel IS
BEGIN
    PROCESS ( s, w )
    BEGIN
        CASE s IS
            WHEN "00" =>
                y <= w ;
            WHEN "01" =>
                y <= w ROR 1 ;
            WHEN "10" =>
                y <= w ROR 2 ;
            WHEN OTHERS =>
                y <= w ROR 3 ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 4.56 VHDL code for the barrel shifter.

Example 4.32

The concept of *parity* is widely used in digital systems for error-checking purposes. When digital information is transmitted from one point to another, perhaps by long wires, it is possible for some bits to become corrupted during the transmission process. For example, the sender may transmit a bit whose value is equal to 1, but the receiver observes a bit whose value is 0. Suppose that a data item consists of n bits. A simple error-checking mechanism can be implemented by including an extra bit, p, which indicates the parity of the n-bit item. Two kinds of parity can be used. For *even parity* the p bit is given the value such that the total number of 1s in the n+1 transmitted bits (comprising the n-bit data and the parity bit p) is even. For *odd parity* the p bit is given the value that makes the total number of 1s odd. The sender generates the p bit based on the n-bit data item that is

to be transmitted. The receiver checks whether the parity of the received item is correct.

Parity generating and checking circuits can be realized with XOR gates. For example, for a four-bit data item consisting of bits $x_3x_2x_1x_0$, the even parity bit can be generated as

$$p = x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

At the receiving end the checking is done using

$$c = p \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

If $c = 0$, then the received item shows the correct parity. If $c = 1$, then an error has occurred. Note that observing $c = 0$ is not a guarantee that the received item is correct. If two or any even number of bits have their values inverted during the transmission, the parity of the data item will not be changed; hence the error will not be detected. But if an odd number of bits are corrupted, then the error will be detected.

Problem: The ASCII code, discussed in Section 1.5.3, uses seven-bit patterns to represent characters. In computer applications it is common to use one byte per character. The eighth bit, b_7, is usually set to 0 for use in digital processing. But, if the character data is to be transmitted from one digital system to another, it may be prudent to use bit b_7 as a parity bit. Write VHDL code that specifies a circuit that accepts an input byte (where $b_7 = 0$) and produces an output byte where b_7 is the even parity bit.

Solution: Let X and Y be the input and output bytes, respectively. Then, the desired solution is given in Figure 4.57.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY parity IS
    PORT ( X  : IN   STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Y  : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END parity ;

ARCHITECTURE Behavior OF parity IS
    SIGNAL p : STD_LOGIC;
BEGIN
    p <= X(0) XOR X(1) XOR X(2) XOR X(3) XOR X(4) XOR X(5) XOR X(6) ;
    Y <= p & X(6 DOWNTO 0) ;
END Behavior ;
```

Figure 4.57 VHDL code for Example 4.32.

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

4.1 Show how the function $f(w_1,w_2,w_3) = \sum m(0, 2, 3, 4, 5, 7)$ can be implemented using a 3-to-8 binary decoder and an OR gate.

4.2 Show how the function $f(w_1,w_2,w_3) = \sum m(1, 2, 3, 5, 6)$ can be implemented using a 3-to-8 binary decoder and an OR gate.

***4.3** Consider the function $f = \overline{w}_1\overline{w}_3 + w_2\overline{w}_3 + \overline{w}_1 w_2$. Use the truth table to derive a circuit for f that uses a 2-to-1 multiplexer.

4.4 Repeat Problem 4.3 for the function $f = \overline{w}_2\overline{w}_3 + w_1 w_2$.

***4.5** For the function $f(w_1,w_2,w_3) = \sum m(0, 2, 3, 6)$, use Shannon's expansion to derive an implementation using a 2-to-1 multiplexer and any other necessary gates.

4.6 Repeat Problem 4.5 for the function $f(w_1,w_2,w_3) = \sum m(0, 4, 6, 7)$.

4.7 Consider the function $f = \overline{w}_2 + \overline{w}_1\overline{w}_3 + w_1 w_3$. Show how repeated application of Shannon's expansion can be used to derive the minterms of f.

4.8 Repeat Problem 4.7 for $f = w_2 + \overline{w}_1\overline{w}_3$.

4.9 Prove Shannon's expansion theorem presented in Section 4.1.2.

*4.10 Section 4.1.2 shows Shannon's expansion in sum-of-products form. Using the principle of duality, derive the equivalent expression in product-of-sums form.

*4.11 Consider the function $f = \overline{w}_1\overline{w}_2 + \overline{w}_2\overline{w}_3 + w_1w_2w_3$. The cost of this minimal sum-of-products expression is 14, which includes four gates and 10 inputs to the gates. Use Shannon's expansion to derive a multilevel circuit that has a lower cost and give the cost of your circuit.

4.12 Use multiplexers to implement the circuit for stage 0 of the carry-lookahead adder in Figure 3.15 (included in the right-most shaded area).

*4.13 Derive minimal sum-of-products expressions for the outputs *a*, *b*, and *c* of the 7-segment display in Figure 4.21.

4.14 Derive minimal sum-of-products expressions for the outputs *d*, *e*, *f*, and *g* of the 7-segment display in Figure 4.21.

4.15 For the function, *f*, in Example 4.23 perform Shannon's expansion with respect to variables w_1 and w_2, rather than w_1 and w_4. How does the resulting circuit compare with the circuit in Figure 4.47?

4.16 Consider the multiplexer-based circuit illustrated in Figure P4.1. Show how the function $f = w_2\overline{w}_3 + w_1w_3 + \overline{w}_2w_3$ can be implemented using only one instance of this circuit.

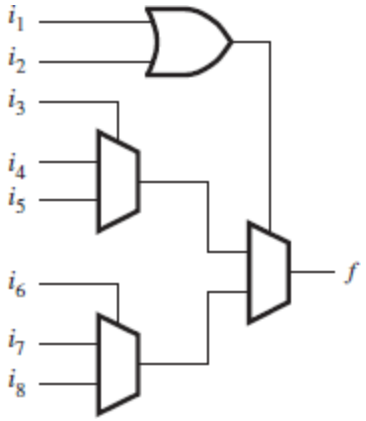


Figure P4.1 A multiplexer-based circuit.

4.17 Show how the function $f = w_1\overline{w}_3 + \overline{w}_1w_3 + w_2\overline{w}_3 + w_1\overline{w}_2$ can be realized using one or more instances of the circuit in Figure P4.1. Note that there are no NOT gates in the circuit; hence complements of

signals have to be generated using the multiplexers in the logic block.

*4.18 Consider the VHDL code in Figure P4.2. What type of circuit does the code represent? Comment on whether or not the style of code used is a good choice for the circuit that it represents.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY problem IS
    PORT ( w              : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En             : IN   STD_LOGIC ;
           y0, y1, y2, y3 : OUT  STD_LOGIC ) ;
END problem ;

ARCHITECTURE Behavior OF problem IS
BEGIN
    PROCESS (w, En)
    BEGIN
        y0 <= '0' ; y1 <= '0' ; y2 <= '0' ; y3 <= '0' ;
        IF En = '1' THEN
            IF w = "00" THEN y0 <= '1' ;
            ELSIF w = "01" THEN y1 <= '1' ;
            ELSIF w = "10" THEN y2 <= '1' ;
            ELSE y3 <= '1' ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure P4.2 Code for Problem 4.18.

4.19 Write VHDL code that represents the function in Problem 6.1, using one selected signal assignment.

4.20 Write VHDL code that represents the function in Problem 6.2, using one selected signal assignment.

4.21 Using a selected signal assignment, write VHDL code for a 4-to-2 binary encoder.

4.22 Using a conditional signal assignment, write VHDL code for an 8-to-3 binary encoder.

4.23 Derive the circuit for an 8-to-3 priority encoder.

4.24 Using a conditional signal assignment, write VHDL code for an 8-to-3 priority encoder.

4.25 Repeat problem 6.24, using an if-then-else statement.

4.26 Create a VHDL entity named *if2to4* that represents a 2-to-4 binary decoder using an if-then-else statement. Create a second entity named *h3to8* that represents the 3-to-8 binary decoder in Figure 4.15, using two instances of the *if2to4* entity.

4.27 Create a VHDL module named *h6to64* that represents a 6-to-64 binary decoder. Use the treelike structure in Figure 4.16, in which the 6-to-64 decoder is built using nine instances of the *h3to8* decoder created in Problem 4.26.

4.28 Write VHDL code for a BCD-to-7-segment code converter, using a selected signal assignment.

4.29 Design a shifter circuit, similar to the one in Figure 4.51, which can shift a four-bit input vector, $W = w_3\ w_2\ w_1\ w_0$, one bit-position to the right when the control signal *Right* is equal to 1, and one bit-position to the left when the control signal *Left* is equal to 1. When *Right* = *Left* = 0, the output of the circuit should be the same as the input vector. Assume that the condition *Right* = *Left* = 1 will never occur.

4.30 Design a circuit that can multiply an eight-bit number, $A = a_7, \ldots, a_0$, by 1, 2, 3, or 4 to produce the result A, $2A$, $3A$, or $4A$, respectively.

4.31 Write VHDL code that implements the task in Problem 4.30.

4.32 Figure 4.48 depicts the relationship between the binary and Gray codes. Design a circuit that can convert Gray code into binary code.

4.33 Example 4.32 and Figure 4.57 show how a circuit that generates an ASCII byte suitable for sending over a communications link may be defined. Write VHDL code for its counterpart at the receiving end, where byte Y (which includes the parity bit) has to be converted into byte X in which the bit x_7 has to be 0. An *error* signal has to be produced, which is set to 0 or 1 depending on whether the parity check indicates correct or erroneous transmission, respectively.

chapter

5

FLIP-FLOPS, REGISTERS, AND COUNTERS

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Logic circuits that can store information
- Flip-flops, which store a single bit
- Registers, which store multiple bits
- Shift registers, which shift the contents of the register
- Counters of various types
- VHDL constructs used to implement storage elements

In previous chapters we considered combinational circuits where the value of each output depends solely on the values of signals applied to the inputs. There exists another class of logic circuits in which the values of the outputs depend not only on the present values of the inputs but also on the past behavior of the circuit. Such circuits include storage elements that store the values of logic signals. The contents of the storage elements are said to represent the *state* of the circuit. When the circuit's inputs change values, the new input values either leave

the circuit in the same state or cause it to change into a new state. Over time the circuit changes through a sequence of states as a result of changes in the inputs. Circuits that behave in this way are referred to as *sequential circuits*.

In this chapter we will introduce circuits that can be used as storage elements. But first, we will motivate the need for such circuits by means of a simple example. Suppose that we wish to control an alarm system, as shown in Figure 5.1. The alarm mechanism responds to the control input $On/\overline{Off}$. It is turned on when $On/\overline{Off} = 1$, and it is off when $On/\overline{Off} = 0$. The desired operation is that the alarm turns on when the sensor generates a positive voltage signal, *Set*, in response to some undesirable event. Once the alarm is triggered, it must remain active even if the sensor output goes back to zero. The alarm is turned off manually by means of a *Reset* input. The circuit requires a memory element to remember that the alarm has to be active until the *Reset* signal arrives.

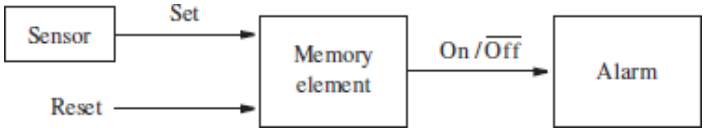


Figure 5.1 Control of an alarm system.

Figure 5.2 gives a rudimentary memory element, consisting of a loop that has two inverters. If we assume that $A = 0$, then $B = 1$. The circuit will maintain these values indefinitely because of the feedback loop. We say that the circuit is in the *state* defined by these values. If we assume that $A = 1$, then $B = 0$, and the circuit will remain in this second state indefinitely. Thus the circuit has two possible states. This circuit is not useful, because it lacks some practical means for changing its state. Useful circuits that exhibit such memory behavior can be constructed with logic gates.

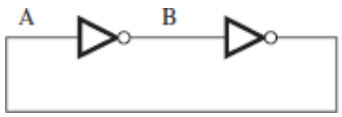


Figure 5.2 A simple memory element.

5.1 Basic Latch

Figure 5.3 presents a memory element built with NOR gates. Its inputs, *Set* and *Reset*, provide the means for changing the state, Q, of the circuit. A more usual way of drawing this circuit is given in Figure 5.4*a*, where the two NOR gates are

said to be connected in *cross-coupled* style. The circuit is referred to as a *basic latch*. Its behavior is described by the table in Figure 5.4*b*. When both inputs R and S are equal to 0 the latch maintains its existing state because of the feedback loop. This state may be either $Q_a = 0$ and $Q_b = 1$, or $Q_a = 1$ and $Q_b = 0$, which is indicated in the table by stating that the Q_a and Q_b outputs have values 0/1 and 1/0, respectively. Observe that Q_a and Q_b are complements of each other in this case. When $R = 1$ and $S = 0$, the latch is *reset* into a state where $Q_a = 0$ and $Q_b = 1$. When $R = 0$ and $S = 1$, the latch is *set* into a state where $Q_a = 1$ and $Q_b = 0$. The fourth possibility is to have $R = S = 1$. In this case both Q_a and Q_b will be 0. The table in Figure 5.4*b* resembles a truth table. However, since it does not represent a combinational circuit in which the values of the outputs are determined solely by the current values of the inputs, it is often called a *characteristic table* rather than a truth table.

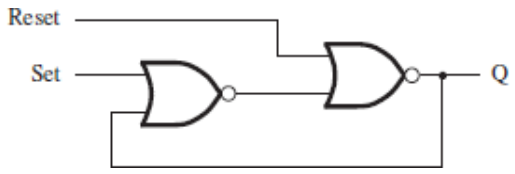


Figure 5.3 A memory element with NOR gates.

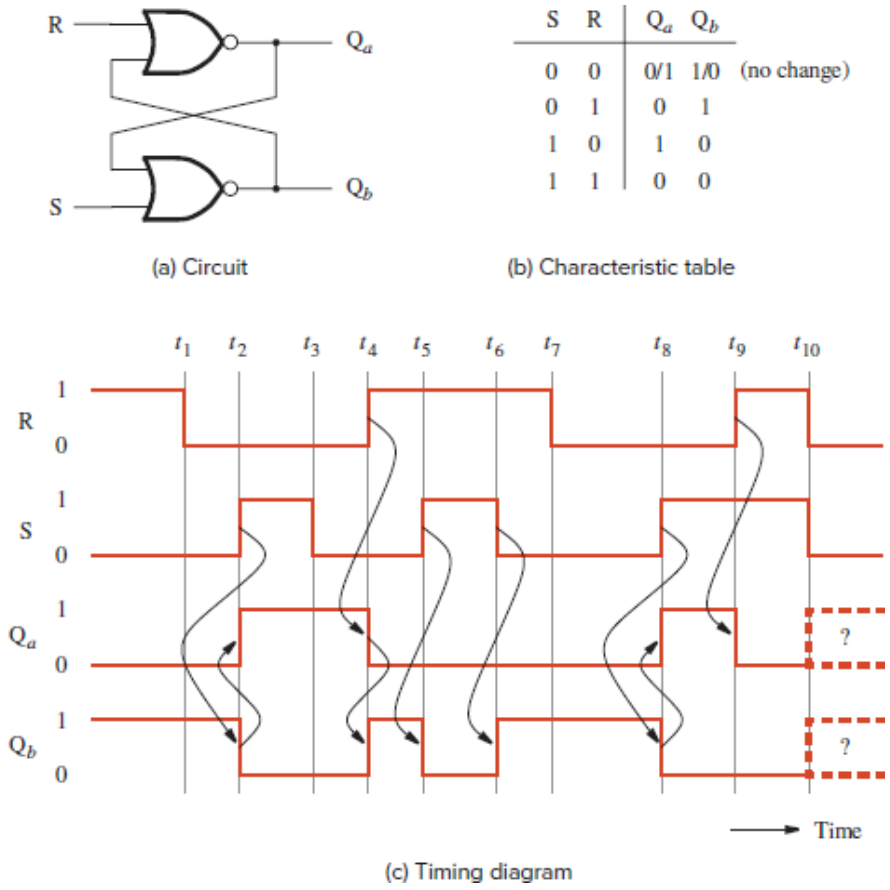


Figure 5.4 A basic latch built with NOR gates.

Figure 5.4*c* gives a timing diagram for the latch, assuming that the propagation delay through the NOR gates is negligible. Of course, in a real circuit the changes in the waveforms would be delayed according to the propagation delays of the gates. We initially reset the latch by applying $R = 1$ and $S = 0$, so that $Q_a = 0$ and $Q_b = 1$. The state of the latch remains unchanged until time t_2, when S becomes equal to 1, causing Q_b to change to 0, which in turn causes Q_a to change to 1 because $\overline{R + Q_b} = 1$. The causality relationship is indicated by the arrows in the diagram. When S goes to 0 at t_3, there is no change in the state because both S and R are then equal to 0. At t_4 we have $R = 1$, which causes Q_a to go to 0, which in turn causes Q_b to go to 1 because $\overline{S + Q_a} = 1$. At t_5 both S and R are equal to 1, which forces both Q_a and Q_b to be equal to 0. As soon as S returns to 0, at t_6, Q_b becomes equal to 1 again. At t_8 we have $S = 1$ and $R = 0$, which causes $Q_b = 0$ and $Q_a = 1$. An interesting situation occurs at t_{10}. From t_9 to t_{10} we have $Q_a = Q_b = 0$ because $R = S = 1$. Now if both R and S change to 0 at t_{10}, both Q_a and Q_b will go to 1. But having both Q_a and Q_b equal to 1 will immediately force $Q_a =$

$Q_b = 0$. There will be an oscillation between $Q_a = Q_b = 0$ and $Q_a = Q_b = 1$. If the delays through the two NOR gates are exactly the same, the oscillation will continue indefinitely. In a real circuit there will invariably be some difference in the delays through these gates, and the latch will eventually settle into one of its two stable states, but we don't know which state it will be. This uncertainty is indicated in the waveforms by dashed lines.

The oscillations discussed above illustrate that even though the basic latch is a simple circuit, careful analysis has to be done to fully appreciate its behavior. In general, any circuit that contains one or more feedback paths, such that the state of the circuit depends on the propagation delays through logic gates, has to be designed carefully. We analyze some of the timing issues related to sequential circuits later in this chapter, and provide more detailed discussions in Chapters 7 and 9.

The latch in Figure 5.4*a* can perform the functions needed for the memory element in Figure 5.1, by connecting the *Set* signal to the *S* input and *Reset* to the *R* input. The Q_a output provides the desired $On/\overline{Off}$ signal. To initialize the operation of the alarm system, the latch is reset. Thus the alarm is off. When the sensor generates the logic value 1, the latch is set and Q_a becomes equal to 1. This turns on the alarm mechanism. If the sensor output returns to 0, the latch retains its state where $Q_a = 1$; hence the alarm remains turned on. The only way to turn off the alarm is by resetting the latch, which is accomplished by making the *Reset* input equal to 1.

5.2 Gated SR Latch

In Section 5.1 we saw that the basic SR latch can serve as a useful memory element. It remembers its state when both the *S* and *R* inputs are 0. It changes its state in response to changes in the signals on these inputs. The state changes occur at the time when the changes in the signals occur. If we cannot control the time of such changes, then we don't know when the latch may change its state.

In the alarm system of Figure 5.1, it may be desirable to be able to enable or disable the entire system by means of a control input, *Enable*. Thus when enabled, the system would function as described above. In the disabled mode, changing the *Set* input from 0 to 1 would not cause the alarm to turn on. The latch in Figure 5.4*a* cannot provide the desired operation. But the latch circuit can be modified to respond to the input signals *S* and *R* only when *Enable* = 1. Otherwise, it would maintain its state.

The modified circuit is depicted in Figure 5.5*a*. It includes two AND gates that provide the desired control. When the control signal *Clk* is equal to 0, the S' and R' inputs to the latch will be 0, regardless of the values of signals *S* and *R*. Hence the latch will maintain its existing state as long as *Clk* = 0. When *Clk* changes to 1, the S' and R' signals will be the same as the *S* and *R* signals, respectively. Therefore, in this mode the latch will behave as we described in Section 5.1. Note that we have used the name *Clk* for the control signal that allows the latch to be set or reset, rather than call it the *Enable* signal. The reason is that such circuits are often used in digital systems where it is desirable to allow the changes in the states of memory elements to occur only at well-defined time intervals, as if they were controlled by a clock. The control signal that defines these time intervals is usually called the *clock* signal. The name *Clk* is meant to reflect this nature of the signal.

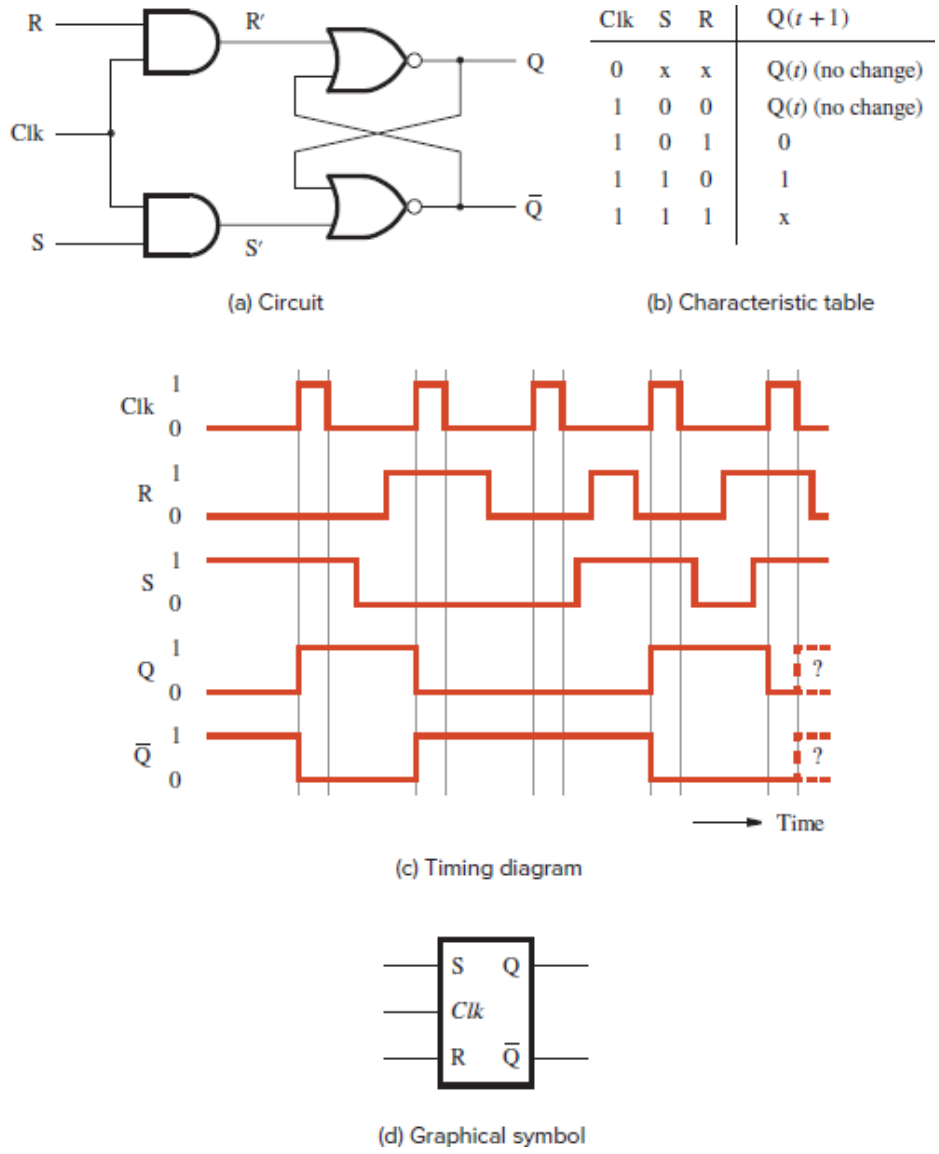


Figure 5.5 Gated SR latch.

Circuits of this type, which use a control signal, are called *gated latches*. Because our circuit exhibits set and reset capability, it is called a *gated SR latch*. Figure 5.5*b* describes its behavior. It defines the state of the Q output at time $t + 1$, namely, $Q(t + 1)$, as a function of the inputs S, R, and Clk. When $Clk = 0$, the latch will remain in the state it is in at time t, that is, $Q(t)$, regardless of the values of inputs S and R. This is indicated by specifying $S = x$ and $R = x$, where x means that it does not matter if the signal value is 0 or 1. When $Clk = 1$, the circuit behaves as the basic latch in Figure 5.4. It is set by $S = 1$ and reset by $R = 1$. The last row of the table, where $S = R = 1$, shows that the state $Q(t + 1)$ is undefined because we don't know whether it will be 0 or 1. This corresponds to the situation described in Section 5.1 in conjunction with the timing diagram in Figure 5.4 at

time t_{10}. At this time both S and R inputs go from 1 to 0, which causes the oscillatory behavior that we discussed. If $S = R = 1$, this situation will occur as soon as *Clk* goes from 1 to 0. To ensure a meaningful operation of the gated SR latch, it is essential to avoid the possibility of having both the S and R inputs equal to 1 when *Clk* changes from 1 to 0.

A timing diagram for the gated SR latch is given in Figure 5.5*c*. It shows *Clk* as a periodic signal that is equal to 1 at regular time intervals to suggest that this is how the clock signal usually appears in a real system. The diagram presents the effect of several combinations of signal values. Observe that we have labeled one output as Q and the other as its complement $\overline{Q}$, rather than Q_a and Q_b as in Figure 5.4. Since the undefined mode, where $S = R = 1$, must be avoided in practice, the normal operation of the latch will have the outputs as complements of each other. Moreover, we will often say that the latch is *set* when Q= 1, and it is *reset* when Q = 0. A graphical symbol for the gated SR latch is given in Figure 5.5*d*.

5.2.1 GATED SR LATCH WITH NAND GATES

So far we have implemented the basic latch with cross-coupled NOR gates. We can also construct the latch with NAND gates. Using this approach, we can implement the gated SR latch as depicted in Figure 5.6. The behavior of this circuit is described by the table in Figure 5.5*b*. Note that in this circuit, the clock is gated by NAND gates, rather than by AND gates. Note also that the S and R inputs are reversed in comparison with the circuit in Figure 5.5*a*.

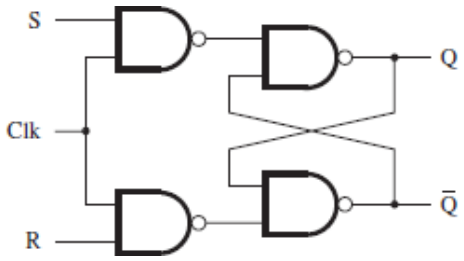


Figure 5.6 Gated SR latch with NAND gates.

5.3 GATED D LATCH

In Section 5.2 we presented the gated SR latch and showed how it can be used as the memory element in the alarm system of Figure 5.1. This latch is useful for many other applications. In this section we describe another gated latch that is

even more useful in practice. It has a single data input, called *D*, and it stores the value on this input, under the control of a clock signal. It is called a *gated D latch*.

To motivate the need for a gated D latch, consider the adder/subtractor unit discussed in Chapter 3 (Figure 3.12). When we described how that circuit is used to add numbers, we did not discuss what is likely to happen with the sum bits that are produced by the adder. Adder/subtractor units are often used as part of a computer. The result of an addition or subtraction operation is often used as an operand in a subsequent operation. Therefore, it is necessary to be able to remember the values of the sum bits generated by the adder until they are needed again. We might think of using the basic latches to remember these bits, one bit per latch. In this context, instead of saying that a latch remembers the value of a bit, it is more illuminating to say that the latch *stores* the value of the bit or simply "stores the bit." We should think of the latch as a storage element.

But can we obtain the desired operation using the basic latches? We can certainly reset all latches before the addition operation begins. Then we would expect that by connecting a sum bit to the *S* input of a latch, the latch would be set to 1 if the sum bit has the value 1; otherwise, the latch would remain in the 0 state. This would work fine if all sum bits are 0 at the start of the addition operation and, after some propagation delay through the adder, some of these bits become equal to 1 to give the desired sum. Unfortunately, the propagation delays that exist in the adder circuit cause a big problem in this arrangement. Suppose that we use a ripple-carry adder. When the *X* and *Y* inputs are applied to the adder, the sum outputs may alternate between 0 and 1 a number of times as the carries ripple through the circuit. The problem is that if we connect a sum bit to the *S* input of a latch, then if the sum bit is temporarily a 1 and then settles to 0 in the final result, the latch will remain set to 1 erroneously.

The problem caused by the alternating values of the sum bits in the adder could be solved by using the gated SR latches, instead of the basic latches. Then we could arrange that the clock signal is 0 during the time needed by the adder to produce a correct sum. After allowing for the maximum propagation delay in the adder circuit, the clock should go to 1 to store the values of the sum bits in the gated latches. As soon as the values have been stored, the clock can return to 0, which ensures that the stored values will be retained until the next time the clock goes to 1. To achieve the desired operation, we would also have to reset all latches to 0 prior to loading the sum-bit values into these latches. This is an awkward way of dealing with the problem, and it is preferable to use the gated D latches instead.

Figure 5.7*a* shows the circuit for a gated D latch. It is based on the gated SR latch, but instead of using the *S* and *R* inputs separately, it has just one data input, *D*. For convenience we have labeled the points in the circuit that are equivalent to

the S and R inputs. If $D = 1$, then $S = 1$ and $R = 0$, which forces the latch into the state Q = 1. If $D = 0$, then $S = 0$ and $R = 1$, which causes Q = 0. Of course, the changes in state occur only when $Clk = 1$.

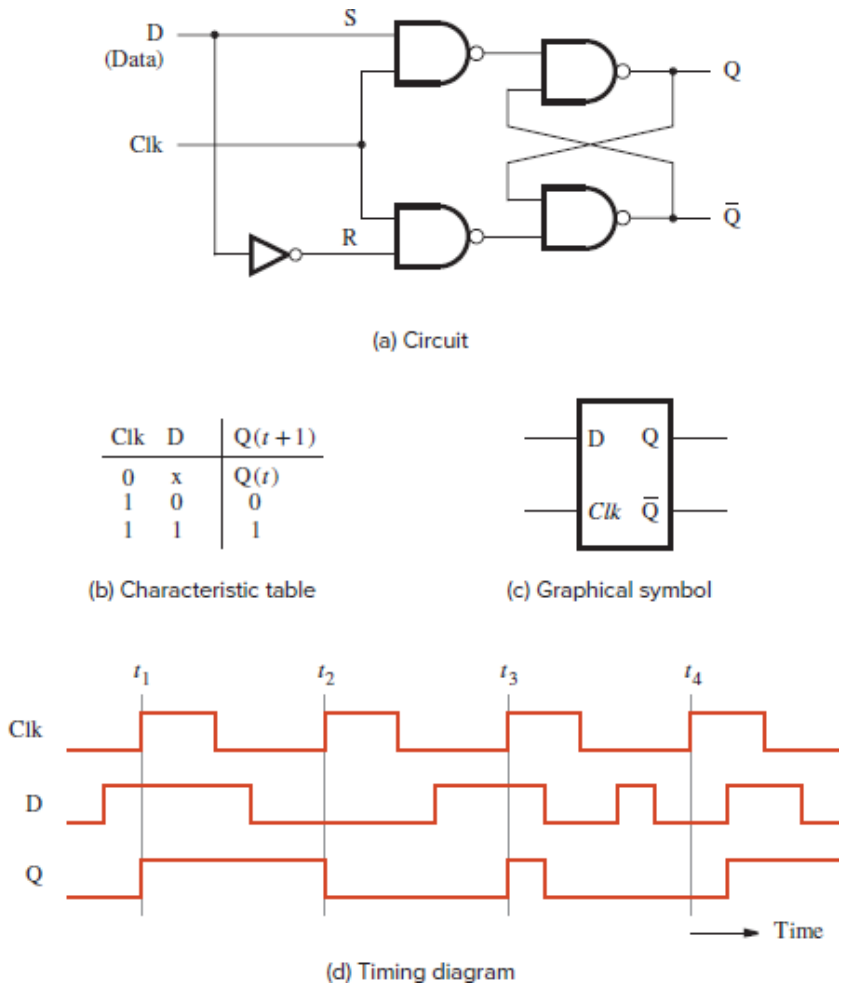


Figure 5.7 Gated D latch.

In this circuit it is impossible to have the troublesome situation where $S = R = 1$. In the gated D latch, the output Q merely tracks the value of the input D while $Clk = 1$. As soon as Clk goes to 0, the state of the latch is frozen until the next time the clock signal goes to 1. Therefore, the gated D latch stores the value of the D input seen at the time the clock changes from 1 to 0. Figure 5.7 also gives the characteristic table, the graphical symbol, and a timing diagram for the gated D latch.

The timing diagram illustrates what happens if the D signal changes while $Clk = 1$. During the third clock pulse, starting at t_3, the output Q changes to 1 because $D = 1$. But midway through the pulse D goes to 0, which causes Q to go to 0. This value of Q is stored when Clk changes to 0. Now no further change in the state of

the latch occurs until the next clock pulse, at t_4. The key point to observe is that as long as the clock has the value 1, the Q output follows the *D* input. But when the clock has the value 0, the Q output cannot change. The logic values are implemented as low and high voltage levels, as explained in detail in Appendix B. Since the output of the gated D latch is controlled by the level of the clock input, the latch is said to be *level sensitive*. The circuits in Figures 5.5 through 5.7 are level sensitive. We will show in Section 5.4 that it is possible to design storage elements for which the output changes only at the point in time when the clock changes from one value to the other. Such circuits are said to be *edge triggered*.

5.3.1 EFFECTS OF PROPAGATION DELAYS

In the previous discussion we ignored the effects of propagation delays. In practical circuits it is essential to take these delays into account. Consider the gated D latch in Figure 5.7*a*. It stores the value of the *D* input that is present at the time the clock signal changes from 1 to 0. It operates properly if the *D* signal is stable (that is, not changing) at the time *Clk* goes from 1 to 0. But it may lead to unpredictable results if the *D* signal also changes at this time. Therefore, the designer of a logic circuit that generates the *D* signal must ensure that this signal is stable when the critical change in the clock signal takes place.

Figure 5.8 illustrates the critical timing region. The minimum time that the *D* signal must be stable prior to the negative edge of the *Clk* signal is called the *setup time*, t_{su}, of the latch. The minimum time that the *D* signal must remain stable after the negative edge of the *Clk* signal is called the *hold time*, t_h, of the latch. The values of t_{su} and t_h depend on the technology used. Manufacturers of integrated circuit chips provide this information on the data sheets that describe their chips. Typical values for a modern technology may be t_{su} = 0.3 ns and t_h = 0.2 ns. We will give examples of how setup and hold times affect the speed of operation of circuits in Section 5.15. The behavior of storage elements when setup or hold times are violated is discussed in Chapter 7.

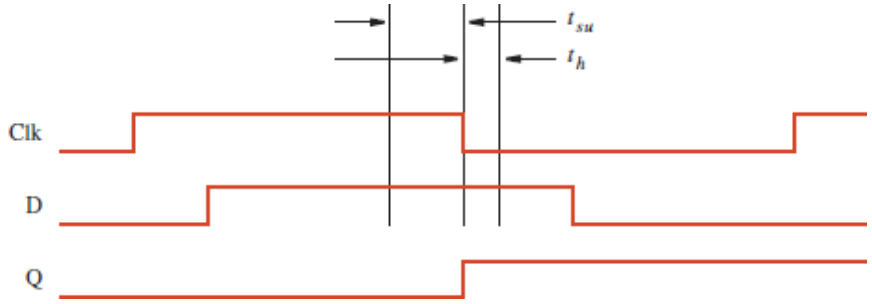


Figure 5.8 Setup and hold times.

5.4 Edge-Triggered D Flip-Flops

In the level-sensitive latches, the state of the latch keeps changing according to the values of input signals during the period when the clock signal is active (equal to 1 in our examples). As we will see in Sections 5.8 and 5.9, there is also a need for storage elements that can change their states no more than once during one clock cycle. We will now discuss circuits that exhibit such behavior.

5.4.1 M-S D Flip-Flop

Consider the circuit given in Figure 5.9*a*, which consists of two gated D latches. The first, called the M-stage, changes its state while *Clock* = 1. The second, called the S-stage changes its state while *Clock* = 0. The operation of the circuit is such that when the clock is high, the M-stage tracks the value of the *D* input signal and the S-stage does not change. Thus the value of Q_m follows any changes in *D*, and the value of Q_s remains constant. When the clock signal changes to 0, the M-stage stops following the changes in the *D* input. At the same time, the S-stage responds to the value of the signal Q_m and changes state accordingly. Since Q_m does not change while *Clock* = 0, the S-stage can undergo at most one change of state during a clock cycle. The S-stage of the M-S circuit changes its state only at the negative-going edge of the clock. The *negative edge* is the edge where the clock signal changes from 1 to 0. Regardless of the number of changes in the *D* input to the M-stage stage during one clock cycle, an observer of the Q_s signal will see only the change that corresponds to the *D* input at the negative edge of the clock.

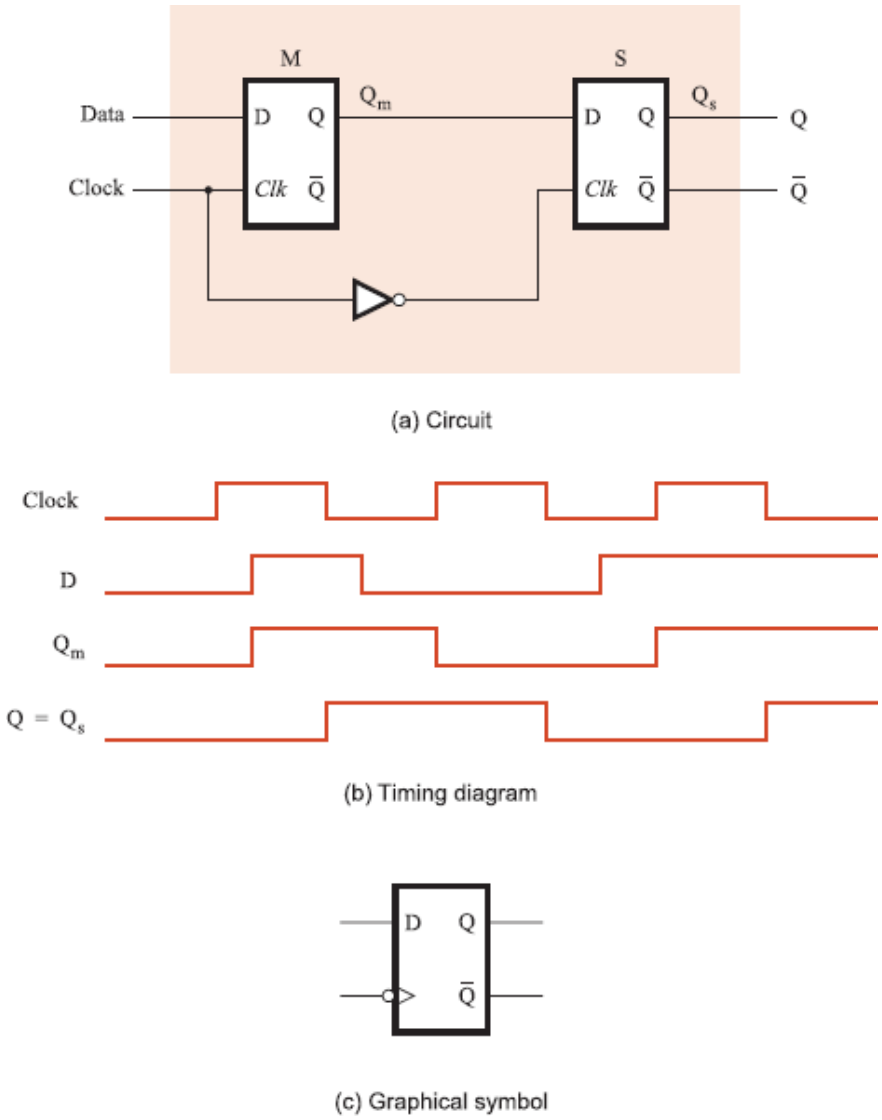


Figure 5.9 M-S D flip-flop.

The circuit in Figure 5.9 is called an *M-S D flip-flop*. The term *flip-flop* denotes a storage element that changes its output state at the edge of a controlling clock signal. The timing diagram for this flip-flop is shown in Figure 5.9*b*. A graphical symbol is given in Figure 5.9*c*. In the symbol we use the > mark to denote that the flip-flop responds to the "active edge" of the clock. We place a bubble on the clock input to indicate that the active edge for this particular circuit is the negative edge.

We can augment the circuit in Figure 5.9*a* by reversing the connections of the clock signal to the M and S stages. That is, include an inverter in the clock input to the M-stage and connect the uncomplemented clock signal to the S-stage. Now, the state of the M-stage will be transferred into the S-stage when the clock signal

changes from 0 to 1. This circuit behaves as a *positive-edge-triggered M-S D flip-flop*. It can be represented by the graphical symbol in Figure 5.9*c* where the bubble on the clock input is removed.

Level-Sensitive versus Edge-Triggered Storage Elements

At this point it is useful to compare the timing in the various storage elements that we have discussed. Figure 5.10 shows three different types of storage elements that are driven by the same data and clock inputs. The first element is a gated D latch, which is level sensitive. The second one is a positive-edge-triggered D flip-flop, and the third one is a negative-edge-triggered D flip-flop. To accentuate the differences between these storage elements, the *D* input changes its values more than once during each half of the clock cycle. Observe that the gated D latch follows the *D* input as long as the clock is high. The positive edge-triggered flip-flop responds only to the value of *D* when the clock changes from 0 to 1. The negative-edge-triggered flip-flop responds only to the value of *D* when the clock changes from 1 to 0.

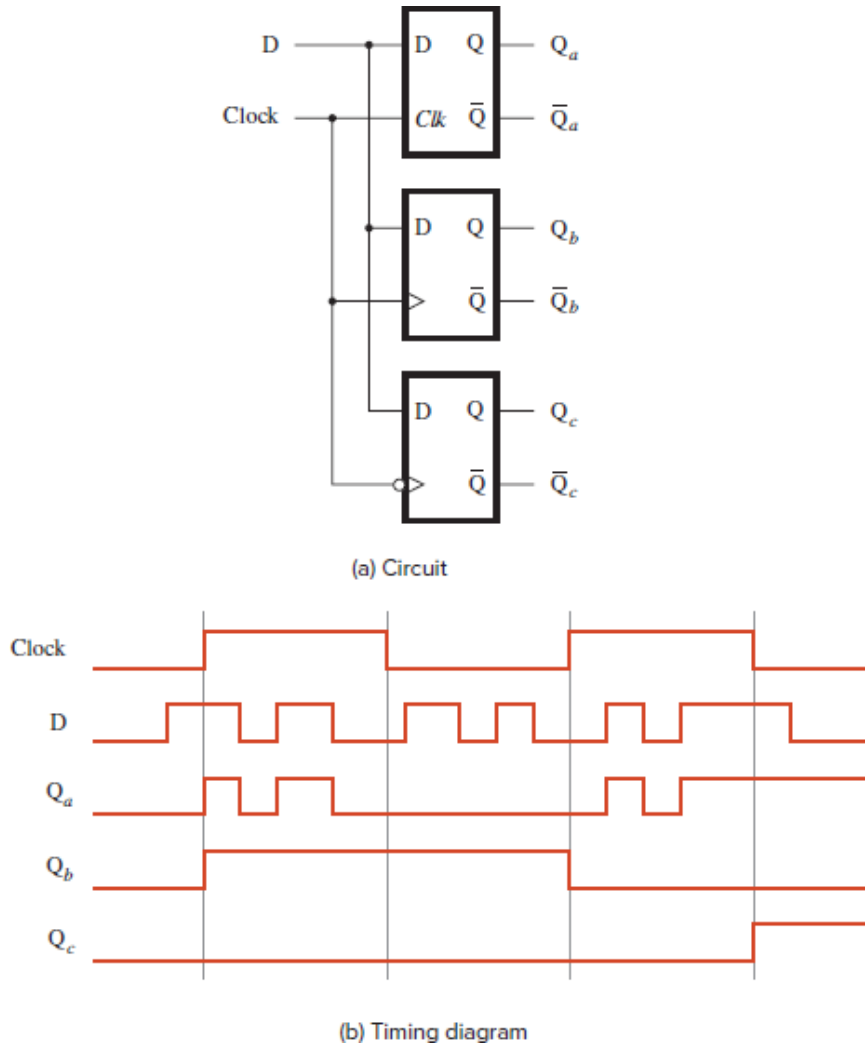


Figure 5.10 Comparison of level-sensitive and edge-triggered D storage elements.

5.4.2 OTHER TYPES OF EDGE-TRIGGERED D FLIP-FLOPS

M-S flip-flops illustrate the concept of edge triggering very clearly. Other circuits have been used to accomplish the same task. Consider the circuit presented in Figure 5.11*a*. It requires only six NAND gates and, hence, has a lower cost than the M-S circuit. The operation of the circuit is as follows. When *Clock* = 0, the outputs of gates 2 and 3 are high. Thus $P1 = P2 = 1$, which maintains the output latch, comprising gates 5 and 6, in its present state. At the same time, the signal $P3$ is equal to D, and $P4$ is equal to its complement $\overline{D}$. When *Clock* changes to 1, the following changes take place. The values of $P3$ and $P4$ are transmitted through gates 2 and 3 to cause $P1 = \overline{D}$ and $P2 = D$, which sets $Q = D$ and $\overline{Q} = \overline{D}$. To operate reliably, $P3$ and $P4$ must be stable when *Clock* changes from 0 to 1. Hence the setup time of the flip-flop is equal to the delay from the D input

through gates 4 and 1 to $P3$. The hold time is given by the delay through gate 3 because once $P2$ is stable, the changes in D no longer matter.

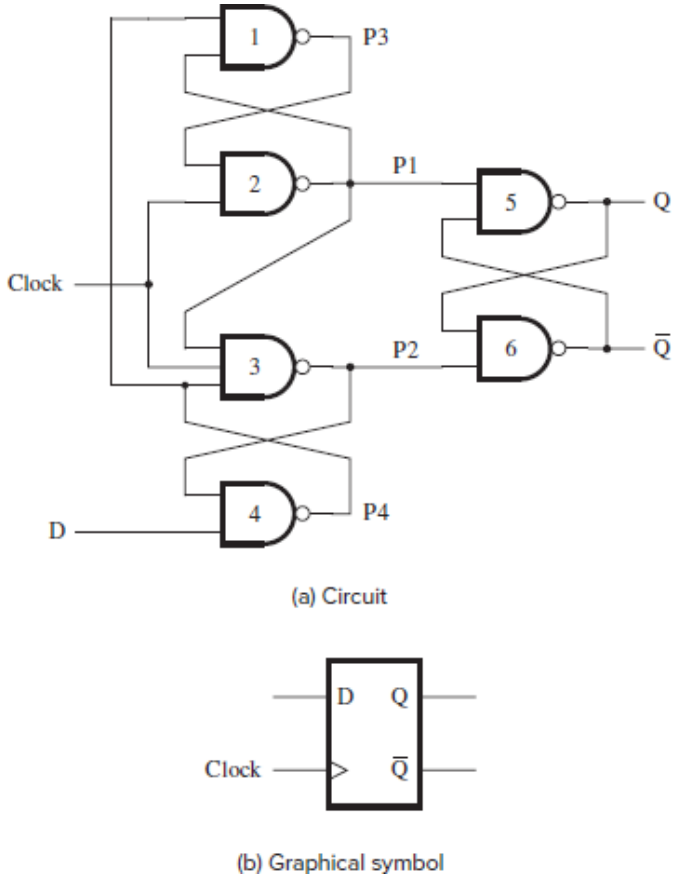


Figure 5.11 A positive-edge-triggered D flip-flop.

For proper operation it is necessary to show that after *Clock* changes to 1 any further changes in D will not affect the output latch as long as *Clock* = 1. We have to consider two cases. Suppose first that $D = 0$ at the positive edge of the clock. Then $P2 = 0$, which will keep the output of gate 4 equal to 1 as long as *Clock* = 1, regardless of the value of the D input. The second case is if $D = 1$ at the positive edge of the clock. Then $P1 = 0$, which forces the outputs of gates 1 and 3 to be equal to 1, regardless of the D input. Therefore, the flip-flop ignores changes in the D input while *Clock* = 1.

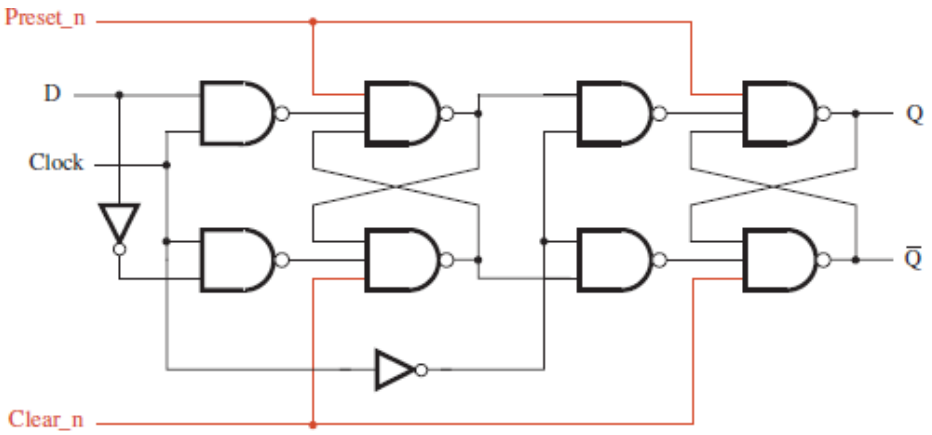
This circuit behaves as a positive-edge-triggered D flip-flop. A similar circuit, constructed with NOR gates, can be used as a negative-edge-triggered flip-flop.

5.4.3 D FLIP-FLOPS WITH CLEAR AND PRESET

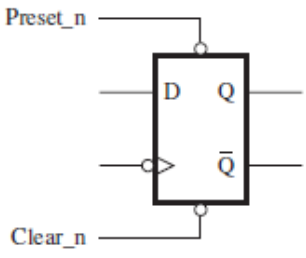
Flip-flops are often used for implementation of circuits that can have many possible states, where the response of the circuit depends not only on the present

values of the circuit's inputs but also on the particular state that the circuit is in at that time. We will discuss a general form of such circuits in Chapter 6. A simple example is a counter circuit that counts the number of occurrences of some event, perhaps the passage of time. We will discuss counters in detail in Section 5.9. A counter comprises a number of flip-flops, whose outputs are interpreted as a number. The counter circuit has to be able to increment or decrement the number. It is also important to be able to force the counter into a known initial state (count). Obviously, it must be possible to clear the count to zero, which means that all flip-flops must have Q = 0. It is equally useful to be able to preset each flip-flop to Q = 1, to insert some specific count as the initial value in the counter. These features can be incorporated into the circuits of Figures 5.9 and 5.11 as follows.

Figure 5.12*a* shows an implementation of the circuit in Figure 5.9*a* using NAND gates. The M-stage is just the gated D latch of Figure 5.7*a*. Instead of using another latch of the same type for the S-stage, we can use the slightly simpler gated SR latch of Figure 5.6. This eliminates one NOT gate from the circuit.



(a) Circuit



(b) Graphical symbol

Figure 5.12 M-S D flip-flop with *Clear* and *Preset*.

A simple way of providing the clear and preset capability is to add an extra input to each NAND gate in the cross-coupled latches, as indicated in red. Placing a 0 on the *Clear_n* input will force the flip-flop into the state Q = 0. If *Clear_n* = 1, then this input will have no effect on the NAND gates. Similarly, *Preset_n* = 0 forces the flip-flop into the state Q = 1, while *Preset_n* = 1 has no effect. To denote that the *Clear_n* and *Preset_n* inputs are active when their value is 0, we appended the letter *n* (for “negative") to these names. We should note that the circuit that uses this flip-flop should not try to force both *Clear_n* and *Preset_n* to 0 at the same time. A graphical symbol for this flip-flop is shown in Figure 5.12*b*.

A similar modification can be done on the edge-triggered flip-flop of Figure 5.11*a*, as indicated in Figure 5.13*a*. Again, both *Clear_n* and *Preset_n* inputs are active low. They do not disturb the flip-flop when they are equal to 1.

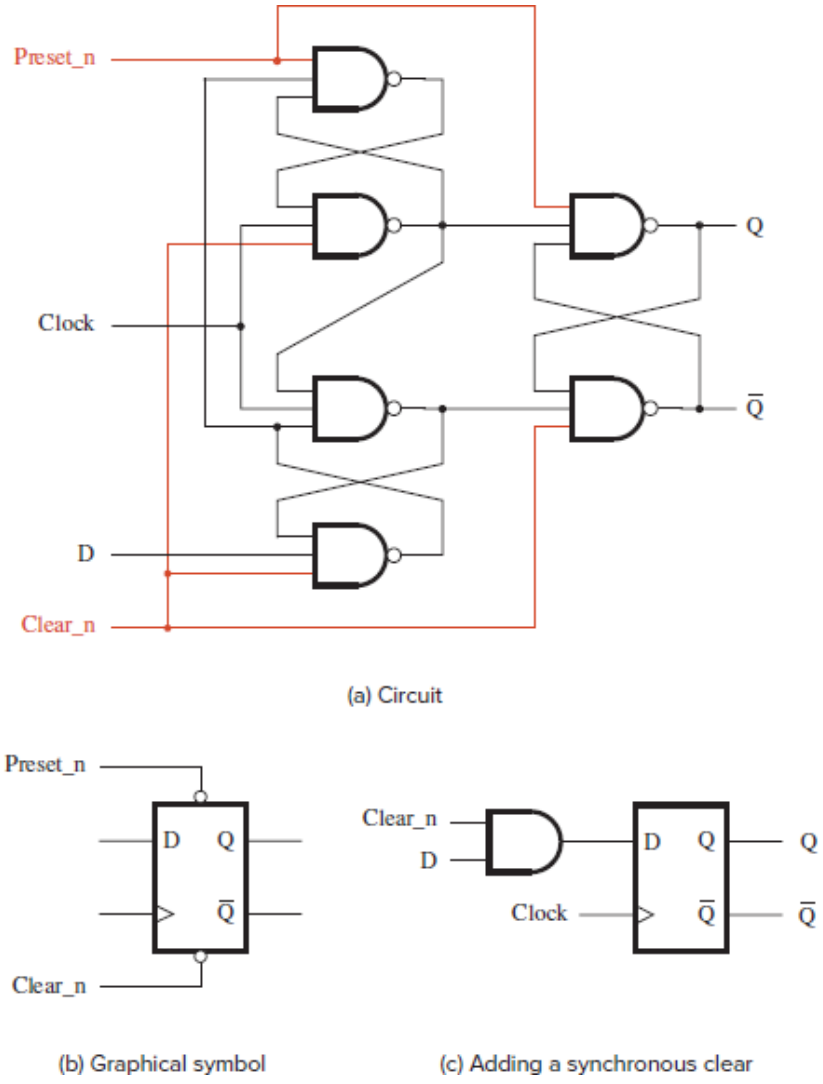


Figure 5.13 Positive-edge-triggered D flip-flop with *Clear* and *Preset*.

In the circuits in Figures 5.12*a* and 5.13*a*, the effect of a low signal on either the *Clear_n* or *Preset_n* input is immediate. For example, if *Clear_n* = 0 then the flip-flop goes into the state Q = 0 immediately, regardless of the value of the clock signal. In such a circuit, where the *Clear_n* signal is used to clear a flip-flop without regard to the clock signal, we say that the flip-flop has an *asynchronous clear*. In practice, it is often preferable to clear the flip-flops on the active edge of the clock. Such *synchronous clear* can be accomplished as shown in Figure 5.13*c*. The flip-flop operates normally when the *Clear_n* input is equal to 1. But if *Clear_n* goes to 0, then on the next positive edge of the clock the flip-flop will be cleared to 0. We will examine the clearing of flip-flops in more detail in Section 5.10.

5.4.4 FLIP-FLOP TIMING PARAMETERS

In section 5.3.1 we discussed timing issues related to latch circuits. In practice such issues are equally important for circuits with flip-flops. Figure 5.14*a* shows a positive-edge triggered flip-flop with asynchronous clear, and part (*b*) of the figure illustrates some important timing parameters for this flip-flop. Data is loaded into the D input of the flip-flop on a positive clock edge, and this logic value must be stable during the setup time, t_{su}, before the clock edge occurs. The data must remain stable during the hold time, t_h, after the edge. If the setup or hold requirements are not adhered to in a circuit that uses this flip-flop, then it may enter an unstable condition known as *metastability*; we discuss this concept in Chapter 7.

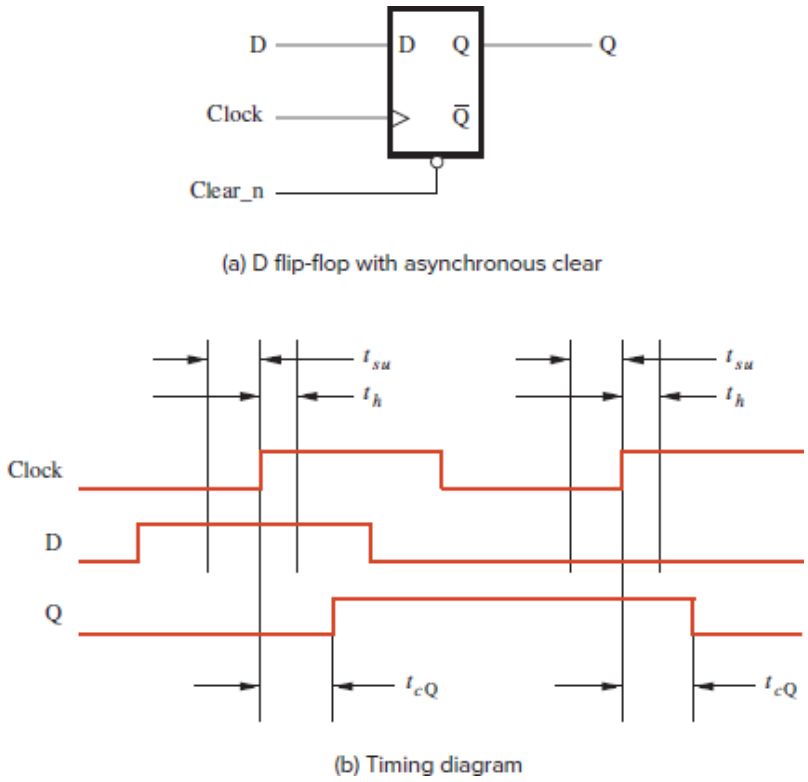


Figure 5.14 Flip-flop timing parameters.

As indicated in Figure 5.14, a clock-to-Q propagation delay, t_{cQ}, is incurred before the value of Q changes after a positive clock edge. In general, the delay may not be exactly the same for the cases when Q changes from 1 to 0 or 0 to 1, but we assume for simplicity that these delays are equal. For the flip-flops in a commercial chip, two values are usually specified for t_{cQ}, representing the maximum and minimum delays that may occur in practice. Specifying a range of values when estimating the delays in a chip is a common practice due to many sources of variation in delay that are caused by the chip manufacturing process. In Section 5.15 we provide some examples that illustrate the effects of flip-flop timing parameters on the operation of circuits.

5.5 T Flip-Flop

The D flip-flop is a versatile storage element that can be used for many purposes. By including some simple logic circuitry to drive its input, the D flip-flop may appear to be a different type of storage element. An interesting modification is presented in Figure 5.15*a*. This circuit uses a positive-edge-triggered D flip-flop.

The *feedback* connections make the input signal D equal to either the value of Q or $\overline{Q}$ under the control of the signal that is labeled T. On each positive edge of the clock, the flip-flop may change its state Q(t). If $T = 0$, then D = Q and the state will remain the same, that is, Q(t+1) = Q(t). But if $T = 1$, then $D = \overline{Q}$ and the new state will be $Q(t+1) = \overline{Q}(t)$. Therefore, the overall operation of the circuit is that it retains its present state if $T = 0$, and it inverts its present state if $T = 1$.

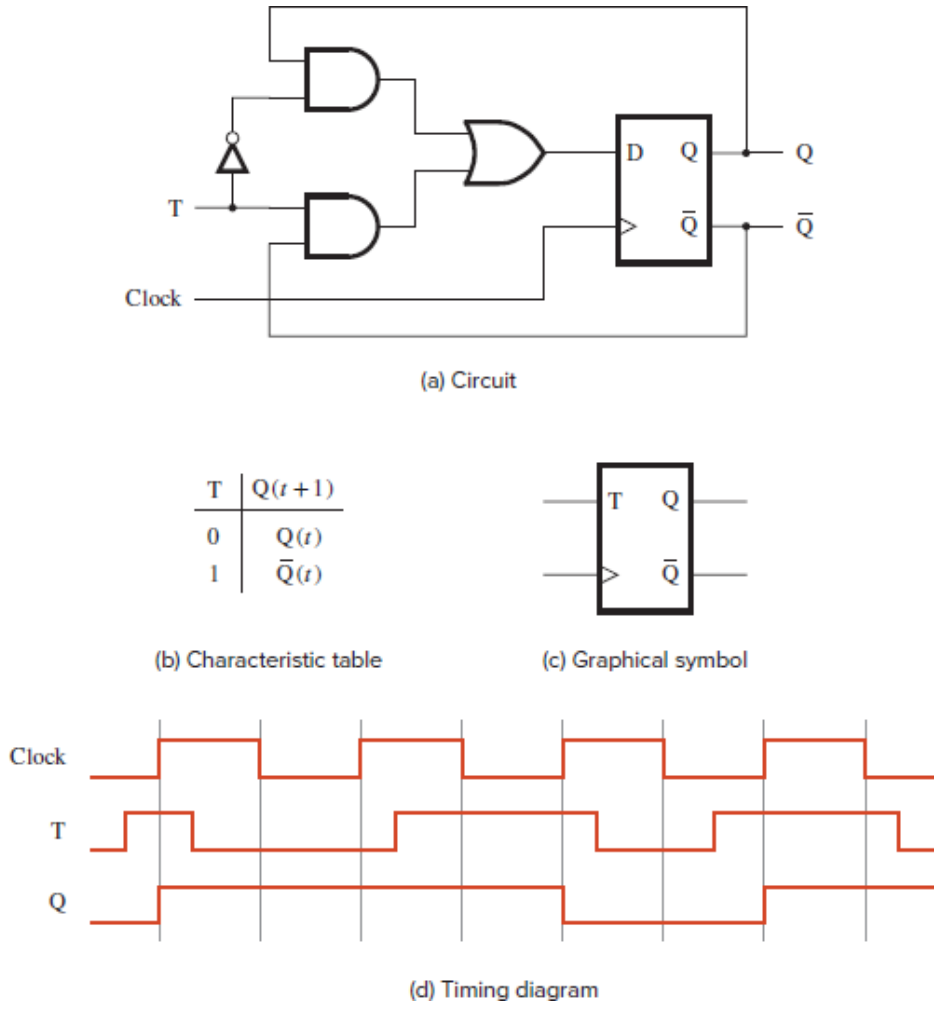


Figure 5.15 T flip-flop.

The operation of the circuit is specified in the form of a characteristic table in Figure 5.15*b*. Any circuit that implements this table is called a *T flip-flop*. The name T flip-flop derives from the behavior of the circuit, which "toggles" its state when $T = 1$. The toggle feature makes the T flip-flop a useful element for building counter circuits, as we will see in Section 5.9.

5.6 JK Flip-Flop

Another interesting circuit can be derived from Figure 5.15*a*. Instead of using a single control input, *T*, we can use two inputs, *J* and *K*, as indicated in Figure 5.16*a*. For this circuit the input *D* is defined as

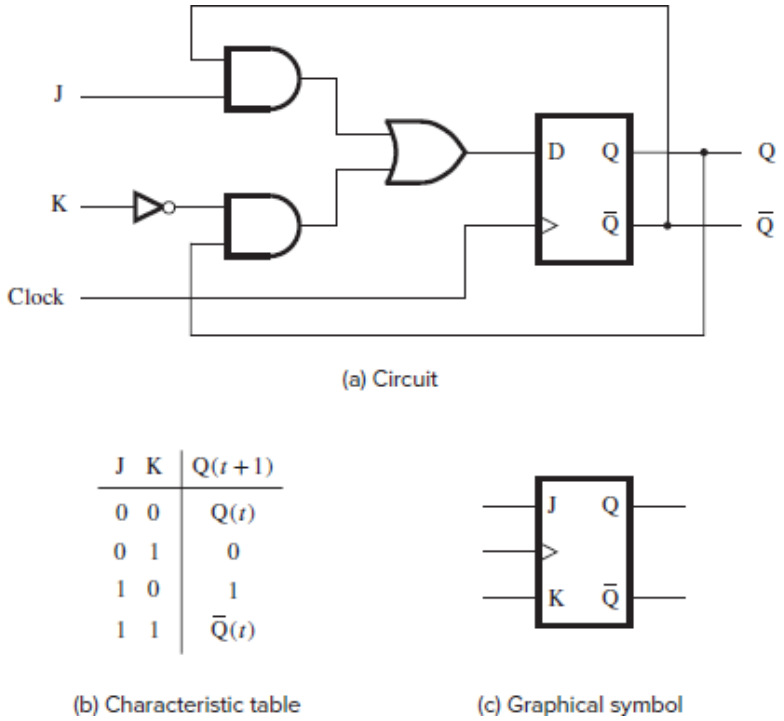


Figure 5.16 JK flip-flop.

$$D = J\overline{Q} + \overline{K}Q$$

A corresponding characteristic table is given in Figure 5.16*b*. The circuit is called a *JK flip-flop*. It combines the behaviors of SR and T flip-flops in a useful way. It behaves as the SR flip-flop, where $J = S$ and $K = R$, for all input values except $J = K = 1$. For the latter case, which has to be avoided in the SR flip-flop, the JK flip-flop toggles its state like the T flip-flop.

The JK flip-flop is a versatile circuit. It can be used for straight storage purposes, just like the D and SR flip-flops. But it can also serve as a T flip-flop by connecting the *J* and *K* inputs together.

5.7 Summary of Terminology

We have used the terminology that is quite common. But the reader should be aware that different interpretations of the terms *latch* and *flip-flop* can be found in the literature. Our terminology can be summarized as follows:

Basic latch is a feedback connection of two NOR gates or two NAND gates, which can store one bit of information. It can be set to 1 using the *S* input and reset to 0 using the *R* input.

Gated latch is a basic latch that includes input gating and a control input signal. The latch retains its existing state when the control input is equal to 0. Its state may be changed when the control signal is equal to 1. In our discussion we referred to the control input as the clock. We considered two types of gated latches:

- **Gated SR latch** uses the *S* and *R* inputs to set the latch to 1 or reset it to 0, respectively.
- **Gated D latch** uses the *D* input to force the latch into a state that has the same logic value as the *D* input.

Flip-flop is a storage element that can have its output state changed only on the edge of the controlling clock signal. If the state changes when the clock signal goes from 0 to 1, we say that the flip-flop is **positive-edge triggered.** If the state changes when the clock signal goes from 1 to 0, we say that the flip-flop is **negative-edge triggered.**

5.8 REGISTERS

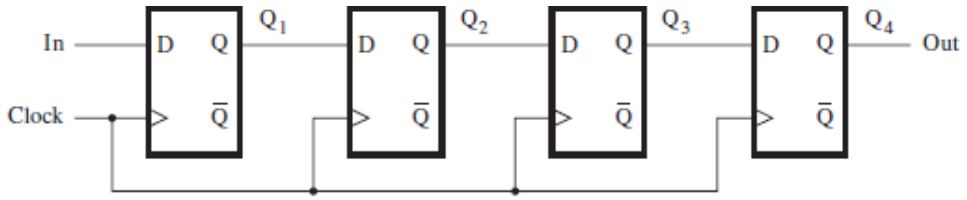
A flip-flop stores one bit of information. When a set of *n* flip-flops is used to store *n* bits of information, such as an *n*-bit number, we refer to these flip-flops as a *register*. A common clock is used for each flip-flop in a register, and each flip-flop operates as described in the previous sections. The term register is merely a convenience for referring to *n*-bit structures consisting of flip-flops.

5.8.1 SHIFT REGISTER

In Section 3.6 we explained that a given number is multiplied by 2 if its bits are shifted one bit position to the left and a 0 is inserted as the new least-significant bit. Similarly, the number is divided by 2 if the bits are shifted one bit-position to

the right. A register that provides the ability to shift its contents is called a *shift register*.

Figure 5.17*a* shows a four-bit shift register that is used to shift its contents one bit-position to the right. The data bits are loaded into the shift register in a serial fashion using the *In* input. The contents of each flip-flop are transferred to the next flip-flop at each positive edge of the clock. An illustration of the transfer is given in Figure 5.17*b*, which shows what happens when the signal values at *In* during eight consecutive clock cycles are 1, 0, 1, 1, 1, 0, 0, and 0, assuming that the initial state of all flip-flops is 0.



(a) Circuit

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

(b) A sample sequence

Figure 5.17 A simple shift register.

To implement a shift register it is necessary to use flip-flops. The level-sensitive gated latches are not suitable, because a change in the value of *In* would propagate through more than one latch during the time when the clock is equal to 1.

5.8.2 PARALLEL-ACCESS SHIFT REGISTER

In computer systems it is often necessary to transfer *n*-bit data items from one location to another. This may be done by transmitting all bits at once using *n*

separate wires, in which case we say that the transfer is performed in *parallel*. But it is also possible to transfer all bits using a single wire, by performing the transfer one bit at a time, in *n* consecutive clock cycles. We refer to this scheme as *serial* transfer. To transfer an *n*-bit data item serially, we can use a shift register that can be loaded with all *n* bits in parallel (in one clock cycle). Then during the next *n* clock cycles, the contents of the register can be shifted out for serial transfer. The reverse operation is also needed. If bits are received serially, then after *n* clock cycles the contents of the register can be accessed in parallel as an *n*-bit item.

Figure 5.18 shows a four-bit shift register that provides the parallel access. A 2-to-1 multiplexer on its *D* input allows each flip-flop to be connected to two different sources. One source is the preceding flip-flop, which is needed for the shift-register operation. The other source is the external input that corresponds to the bit that is to be loaded into the flip-flop as a part of the parallel-load operation. The control signal $\overline{Shift}/Load$ is used to select the mode of operation. If $\overline{Shift}/Load = 0$, then the circuit operates as a shift register. If $\overline{Shift}/Load = 1$, then the parallel input data are loaded into the register. In both cases the action takes place on the positive edge of the clock.

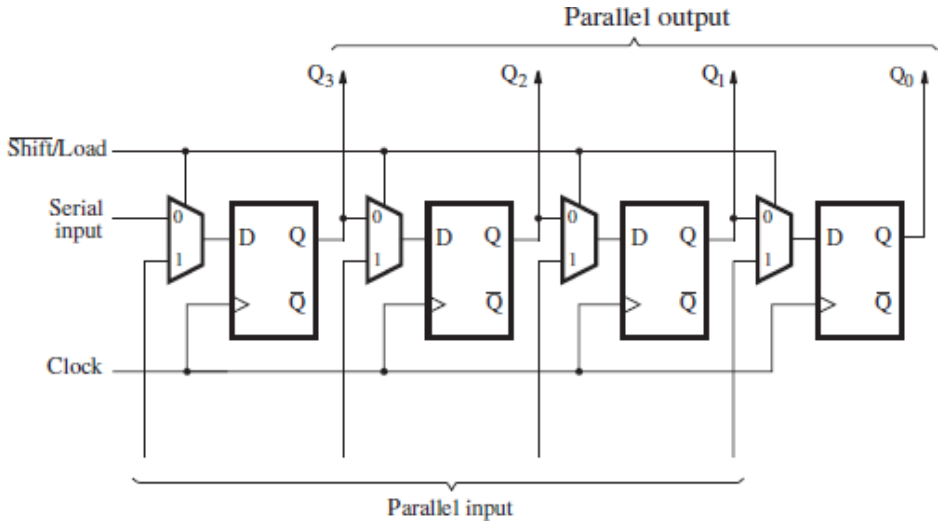


Figure 5.18 Parallel-access shift register.

In Figure 5.18 we have chosen to label the flip-flops' outputs as Q_3, …, Q_0 because shift registers are often used to hold binary numbers. The contents of the register can be accessed in parallel by observing the outputs of all flip-flops. The flip-flops can also be accessed serially, by observing the values of Q_0 during consecutive clock cycles while the contents are being shifted. A circuit in which data can be loaded in series and then accessed in parallel is called a series-to-

parallel converter. Similarly, the opposite type of circuit is a parallel-to-series converter. The circuit in Figure 5.18 can perform both of these functions.

5.9 COUNTERS

In Chapter 3 we dealt with circuits that perform arithmetic operations. We showed how adder/subtractor circuits can be designed, either using a simple cascaded (ripple-carry) structure that is inexpensive but slow or using a more complex carry-lookahead structure that is both more expensive and faster. In this section we examine special types of addition and subtraction operations, which are used for the purpose of counting. In particular, we want to design circuits that can increment or decrement a count by 1. Counter circuits are used in digital systems for many purposes. They may count the number of occurrences of certain events, generate timing intervals for control of various tasks in a system, keep track of time elapsed between specific events, and so on.

Counters can be implemented using the adder/subtractor circuits discussed in Chapter 3 and the registers discussed in Section 5.8. However, since we only need to change the contents of a counter by 1, it is not necessary to use such elaborate circuits. Instead, we can use much simpler circuits that have a significantly lower cost. We will show how the counter circuits can be designed using T and D flip-flops.

5.9.1 ASYNCHRONOUS COUNTERS

The simplest counter circuits can be built using T flip-flops because the toggle feature is naturally suited for the implementation of a counting operation.

Up-Counter with T Flip-Flops

Figure 5.19*a* gives a three-bit counter capable of counting from 0 to 7. The clock inputs of the three flip-flops are connected in cascade. The *T* input of each flip-flop is connected to a constant 1, which means that the state of the flip-flop will be reversed (toggled) at each positive edge of its clock. We are assuming that the purpose of this circuit is to count the number of pulses that occur on the primary input called *Clock*. Thus the clock input of the first flip-flop is connected to the *Clock* signal. The other two flip-flops have their clock inputs driven by the $\overline{Q}$ output of the preceding flip-flop. Therefore, they toggle their state whenever the preceding flip-flop changes its state from Q = 1 to Q = 0, which results in a positive edge of the $\overline{Q}$ signal.

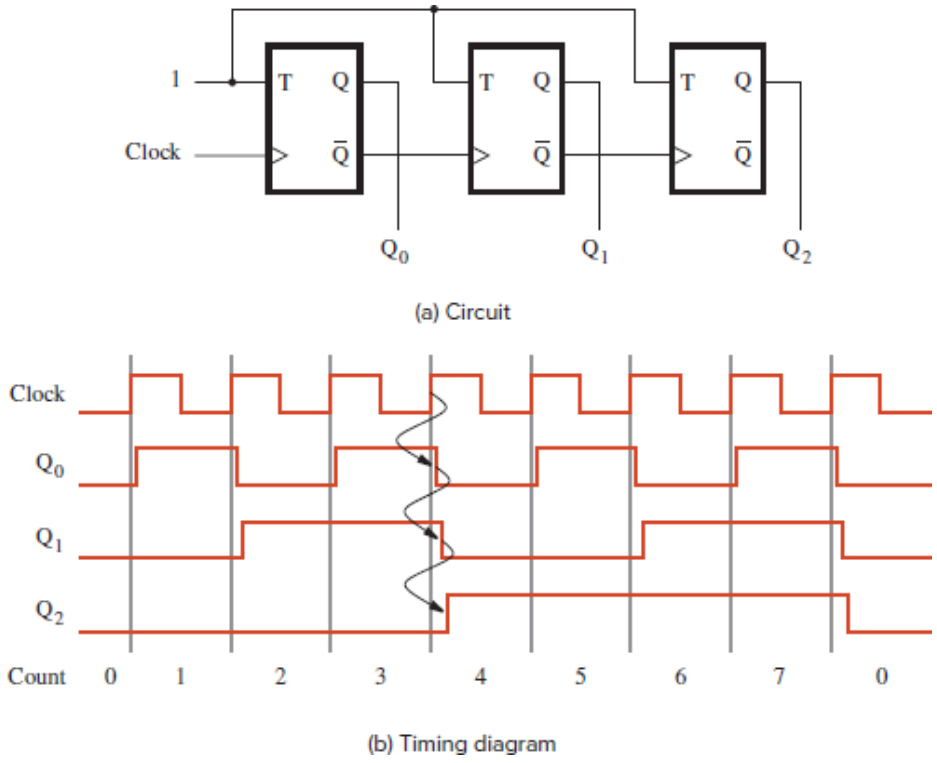


Figure 5.19 A three-bit up-counter.

Figure 5.19*b* shows a timing diagram for the counter. The value of Q_0 toggles once each clock cycle. The change takes place after a short delay following the positive edge of the *Clock* signal. The delay is caused by the propagation time through the flip-flop. Since the second flip-flop is clocked by $\overline{Q}_0$, the value of Q_1 changes shortly after the negative edge of the Q_0 signal. Similarly, the value of Q_2 changes shortly after the negative edge of the Q_1 signal. If we look at the values $Q_2Q_1Q_0$ as the count, then the timing diagram indicates that the counting sequence is 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, and so on. This circuit is a modulo-8 counter. Because it counts in the upward direction, we call it an *up-counter*.

The counter in Figure 5.19*a* has three *stages*, each comprising a single flip-flop. Only the first stage responds directly to the *Clock* signal; we say that this stage is *synchronized* to the clock. The other two stages respond after an additional delay. For example, when *Count* = 3, the next clock pulse will cause the *Count* to go to 4. As indicated by the arrows in the timing diagram in Figure 5.19*b*, this change requires the toggling of the states of all three flip-flops. The change in Q_0 is observed only after a propagation delay from the positive edge of *Clock*. The Q_1 and Q_2 flip-flops have not yet changed; hence for a brief time the count is $Q_2Q_1Q_0 = 010$. The change in Q_1 appears after a second propagation

delay, at which point the count is 000. Finally, the change in Q_2 occurs after a third delay, at which point the stable state of the circuit is reached and the count is 100. This behavior is similar to the rippling of carries in the ripple-carry adder circuit of Figure 3.5. The circuit in Figure 5.19*a* is an *asynchronous counter*, or a *ripple counter*.

Down-Counter with T Flip-Flops

A slight modification of the circuit in Figure 5.19*a* is presented in Figure 5.20*a*. The only difference is that in Figure 5.20*a* the clock inputs of the second and third flip-flops are driven by the Q outputs of the preceding stages, rather than by the $\overline{Q}$ outputs. The timing diagram, given in Figure 5.20*b*, shows that this circuit counts in the sequence 0, 7, 6, 5, 4, 3, 2, 1, 0, 7, and so on. Because it counts in the downward direction, we say that it is a *down-counter*.

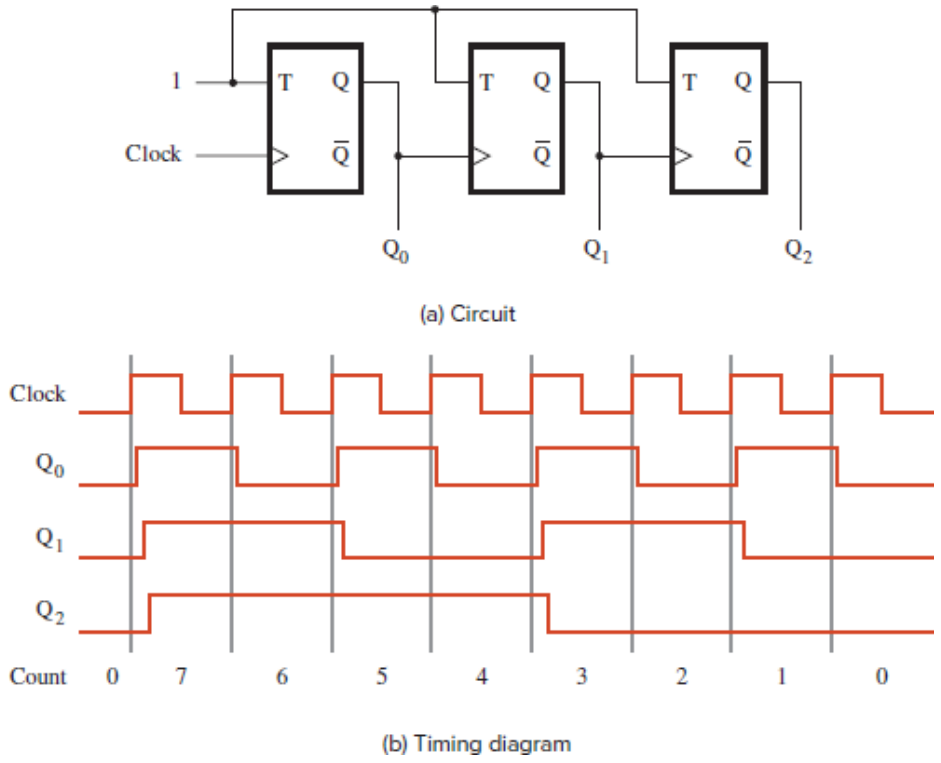


Figure 5.20 A three-bit down-counter.

It is possible to combine the functionality of the circuits in Figures 5.19*a* and 5.20*a* to form a counter that can count either up or down. Such a counter is called an *up/down-counter*. We leave the derivation of this counter as an exercise for the reader (Problem 5.15).

5.9.2 SYNCHRONOUS COUNTERS

The asynchronous counters in Figures 5.19*a* and 5.20*a* are simple, but not very fast. If a counter with a larger number of bits is constructed in this manner, then the delays caused by the cascaded clocking scheme may become too long to meet the desired performance requirements. We can build a faster counter by clocking all flip-flops at the same time, using the approach described below.

Synchronous Counter with T Flip-Flops

Table 5.1 shows the contents of a three-bit up-counter for eight consecutive clock cycles, assuming that the count is initially 0. Observing the pattern of bits in each row of the table, it is apparent that bit Q_0 changes on each clock cycle. Bit Q_1 changes only when $Q_0 = 1$. For example, Q_1 changes in clock cycle 2 because $Q_0 = 1$ in clock cycle 1. Bit Q_2 changes only when both Q_1 and Q_0 are equal to 1. For instance, Q_2 changes in clock cycle 4 because $Q_1 = Q_0 = 1$ in clock cycle 3. In general, for an *n*-bit up-counter, a given flip-flop changes its state only when all the preceding flip-flops are in the state Q = 1. Therefore, if we use T flip-flops to realize the counter, then the *T* inputs are defined as

$$
\begin{aligned}
T_0 &= 1 \\
T_1 &= Q_0 \\
T_2 &= Q_0 Q_1 \\
T_3 &= Q_0 Q_1 Q_2 \\
&\vdots \\
T_n &= Q_0 Q_1 \cdots Q_{n-1}
\end{aligned}
$$

Table 5.1 Derivation of the synchronous up-counter.

An example of a four-bit counter based on these expressions is given in Figure 5.21*a*. Instead of using AND gates of increased size for each stage, we use a factored arrangement as shown in the figure. This arrangement does not slow down the response of the counter, because all flip-flops change their states after a propagation delay from the positive edge of the clock. Note that a change in the value of Q_0 may have to propagate through several AND gates to reach the flip-flops in the higher stages of the counter, which requires a certain amount of time. This time must not exceed the clock period. Actually, it must be less than the clock period minus the setup time for the flip-flops.

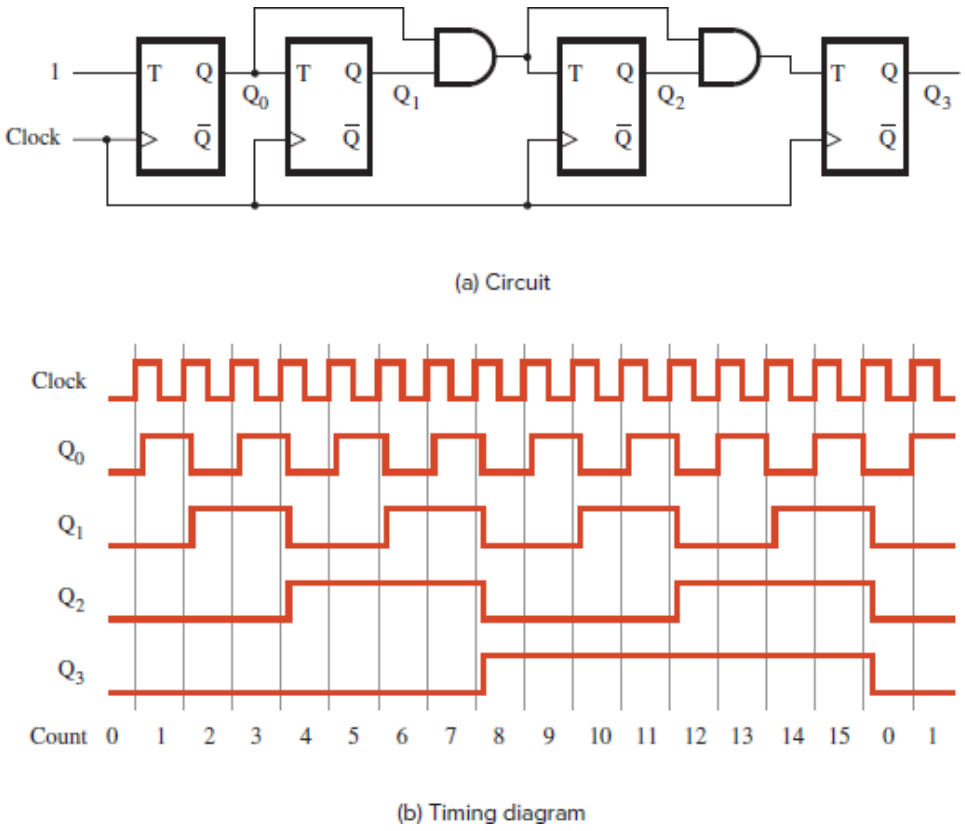


Figure 5.21 A four-bit synchronous up-counter.

Clock cycle	Q_2 Q_1 Q_0
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1
8	0 0 0

Q_1 changes
Q_2 changes

Figure 5.21*b* gives a timing diagram. It shows that the circuit behaves as a modulo-16 up-counter. Since all changes take place with the same delay after the active edge of the *Clock* signal, the circuit is called a *synchronous counter*.

Enable and Clear Capability

The counters in Figures 5.19 through 5.21 change their contents in response to each clock pulse. Often it is desirable to be able to inhibit counting, so that the count remains in its present state. This may be accomplished by including an

Enable control signal, as indicated in Figure 5.22. The circuit is the counter of Figure 5.21, where the *Enable* signal controls directly the *T* input of the first flip-flop. Connecting the *Enable* also to the AND-gate chain means that if *Enable* = 0, then all *T* inputs will be equal to 0. If *Enable* = 1, then the counter operates as explained previously.

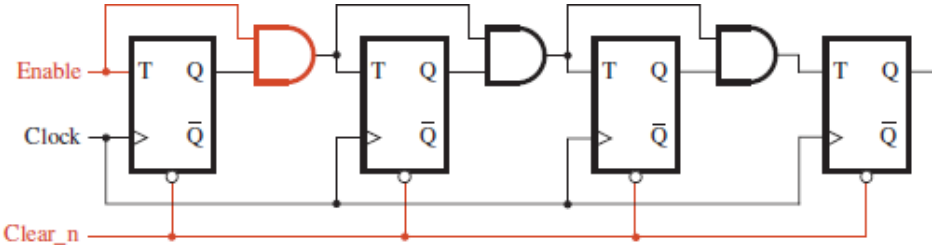


Figure 5.22 Inclusion of Enable and Clear capability.

In many applications it is necessary to start with the count equal to zero. This is easily achieved if the flip-flops can be cleared, as explained in Section 5.4.3. The clear inputs on all flip-flops can be tied together and driven by a *Clear_n* control input.

Synchronous Counter with D Flip-Flops

While the toggle feature makes T flip-flops a natural choice for the implementation of counters, it is also possible to build counters using other types of flip-flops. The JK flip-flops can be used in exactly the same way as the T flip-flops because if the *J* and *K* inputs are tied together, a JK flip-flop becomes a T flip-flop. We will now consider using D flip-flops for this purpose.

It is not obvious how D flip-flops can be used to implement a counter. We will present a formal method for deriving such circuits in Chapter 6. Here we will present a circuit structure that meets the requirements but will leave the derivation for Chapter 6. Figure 5.23 gives a four-bit up-counter that counts in the sequence 0, 1, 2, …, 14, 15, 0, 1, and so on. The count is indicated by the flip-flop outputs $Q_3Q_2Q_1Q_0$. If we assume that *Enable* = 1, then the *D* inputs of the flip-flops are defined by the expressions

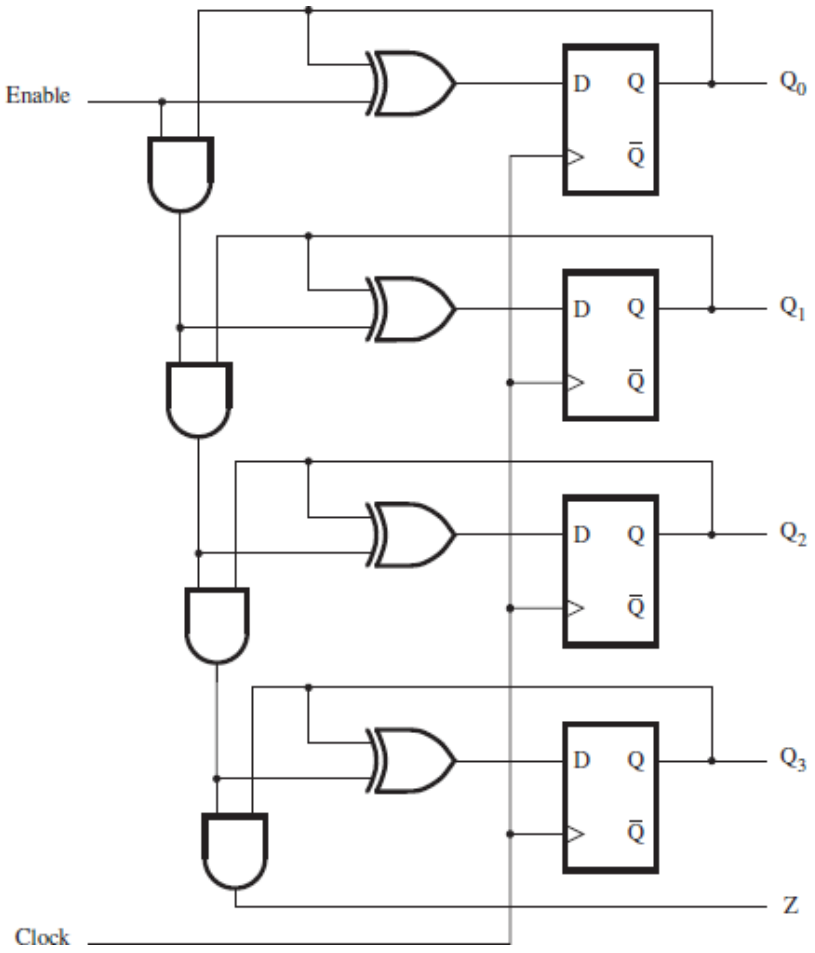


Figure 5.23 A four-bit counter with D flip-flops.

$$D_0 = Q_0 \oplus 1 = \overline{Q}_0$$
$$D_1 = Q_1 \oplus Q_0$$
$$D_2 = Q_2 \oplus Q_1 Q_0$$
$$D_3 = Q_3 \oplus Q_2 Q_1 Q_0$$

For a larger counter the *i*th stage is defined by

$$D_i = Q_i \oplus Q_{i-1} Q_{i-2} \cdots Q_1 Q_0$$

We will show how to derive these equations in Chapter 6.

We have included the *Enable* control signal in Figure 5.23 so that the counter counts the clock pulses only if *Enable* = 1. In effect, the above equations are modified to implement the circuit in the figure as follows

$$D_0 = Q_0 \oplus Enable$$
$$D_1 = Q_1 \oplus Q_0 \cdot Enable$$
$$D_2 = Q_2 \oplus Q_1 \cdot Q_0 \cdot Enable$$
$$D_3 = Q_3 \oplus Q_2 \cdot Q_1 \cdot Q_0 \cdot Enable$$

The operation of the counter is based on our observation for Table 5.1 that the state of the flip-flop in stage *i* changes only if all preceding flip-flops are in the state Q = 1. This makes the output of the AND gate that feeds stage *i* equal to 1, which causes the output of the XOR gate connected to D_i to be equal to $\overline{Q}_i$. Otherwise, the output of the XOR gate provides $D_i = Q_i$, and the flip-flop remains in the same state.

We have included an extra AND gate that produces the output *Z*. This signal makes it easy to concatenate two such counters to create a larger counter. It is also useful in applications where it is necessary to detect the state where the count has reached its maximum value (all 1s) and will go to 0 in the next clock cycle.

The counter in Figure 5.23 is essentially the same as the circuit in Figure 5.22. We showed in Figure 5.15*a* that a T flip-flop can be formed from a D flip-flop by providing the extra gating that gives

$$D = Q\overline{T} + \overline{Q}T$$
$$= Q \oplus T$$

Thus in each stage in Figure 5.23, the D flip-flop and the associated XOR gate implement the functionality of a T flip-flop.

5.9.3 COUNTERS WITH PARALLEL LOAD

Often it is necessary to start counting with the initial count being equal to 0. This state can be achieved by using the capability to clear the flip-flops as indicated in Figure 5.22. But sometimes it is desirable to start with a different count. To allow this mode of operation, a counter circuit must have some inputs through which the initial count can be loaded. Using the *Clear* and *Preset* inputs for this purpose is a possibility, but a better approach is discussed below.

The circuit of Figure 5.23 can be modified to provide the parallel-load capability as shown in Figure 5.24. A two-input multiplexer is inserted before each *D* input. One input to the multiplexer is used to provide the normal counting operation. The other input is a data bit that can be loaded directly into the flip-flop. A control input, *Load*, is used to choose the mode of operation. The circuit counts when *Load* = 0. A new initial value, D_3 D_2 D_1 D_0, is loaded into the counter when *Load* = 1.

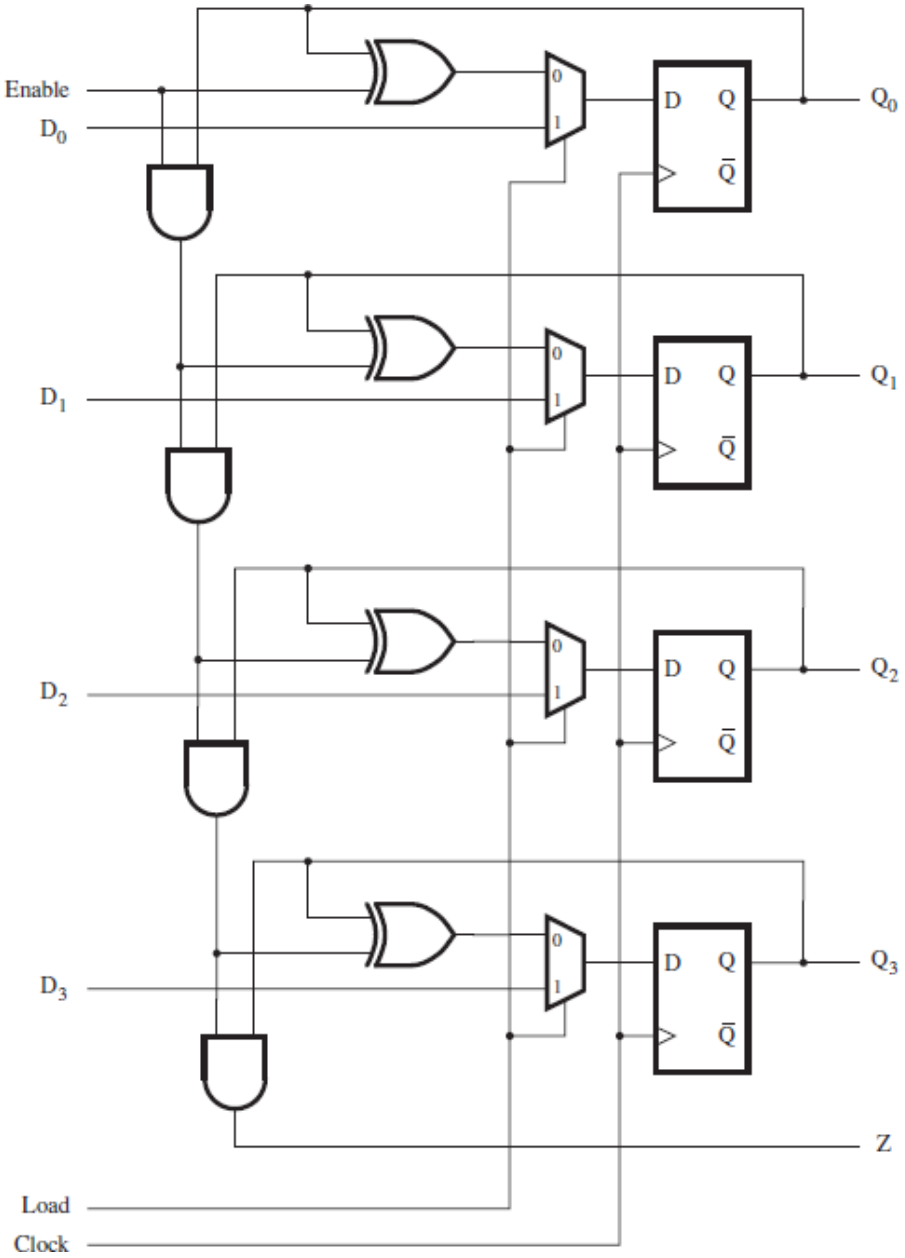


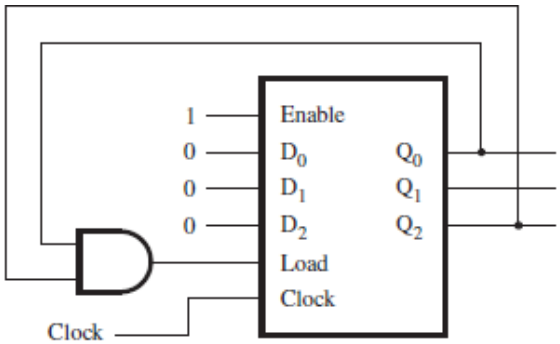
Figure 5.24 A counter with parallel-load capability.

5.10 Reset Synchronization

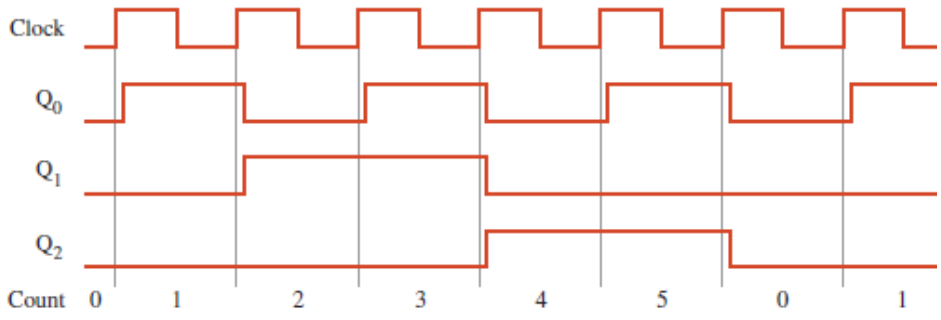
We have already mentioned that it is important to be able to clear, or *reset*, the contents of a counter prior to commencing a counting operation. This can be done using the clear capability of the individual flip-flops. But we may also be interested in resetting the count to 0 during the normal counting process. An *n*-bit

up-counter functions naturally as a modulo-2^n counter. Suppose that we wish to have a counter that counts modulo some base that is not a power of 2. For example, we may want to design a modulo-6 counter, for which the counting sequence is 0, 1, 2, 3, 4, 5, 0, 1, and so on.

The most straightforward approach is to recognize when the count reaches 5 and then reset the counter. An AND gate can be used to detect the occurrence of the count of 5. Actually, it is sufficient to ascertain that $Q_2 = Q_0 = 1$, which is true only for 5 in our desired counting sequence. A circuit based on this approach is given in Figure 5.25*a*. It uses a three-bit synchronous counter of the type depicted in Figure 5.24. The parallel-load feature of the counter is used to reset its contents when the count reaches 5. The resetting action takes place at the positive clock edge after the count has reached 5. It involves loading $D_2\ D_1D_0 = 0\ 0\ 0$ into the flip-flops. As seen in the timing diagram in Figure 5.25*b*, the desired counting sequence is achieved, with each value of the count being established for one full clock cycle. Because the counter is reset on the active edge of the clock, we say that this type of counter has a *synchronous reset*.



(a) Circuit



(b) Timing diagram

Figure 5.25 A modulo-6 counter with synchronous reset.

Consider now the possibility of using the clear feature of individual flip-flops, shown in Figures 5.12*a* and 5.13*a*, rather than the parallel-load approach. The circuit in Figure 5.26*a* illustrates one possibility. It uses the counter structure of Figure 5.21*a*. Since the clear inputs are active when low, a NAND gate is used to detect the occurrence of the count of 5 and cause the clearing of all three flip-flops. Conceptually, this seems to work fine, but closer examination reveals a potential problem. The timing diagram for this circuit is given in Figure 5.26*b*. It shows a difficulty that arises when the count is equal to 5. As soon as the count reaches this value, the NAND gate triggers the resetting action. The flip-flops are cleared to 0 a short time after the NAND gate has detected the count of 5. This time depends on the gate delays in the circuit, but not on the clock. Therefore, signal values $Q_2Q_1Q_0 = 1\ 0\ 1$ are maintained for a time that is much less than a clock cycle. Depending on a particular application of such a counter, this may be adequate, but it may also be completely unacceptable. For example, if the counter is used in a digital system where all operations in the system are synchronized by the same clock, then this narrow pulse denoting *Count* = 5 would not be seen by the rest of the system. This is not a good way of designing circuits, because pulses of short length often cause unforeseen difficulties in practice. The approach employed in Figure 5.26*a* is said to use *asynchronous reset*.

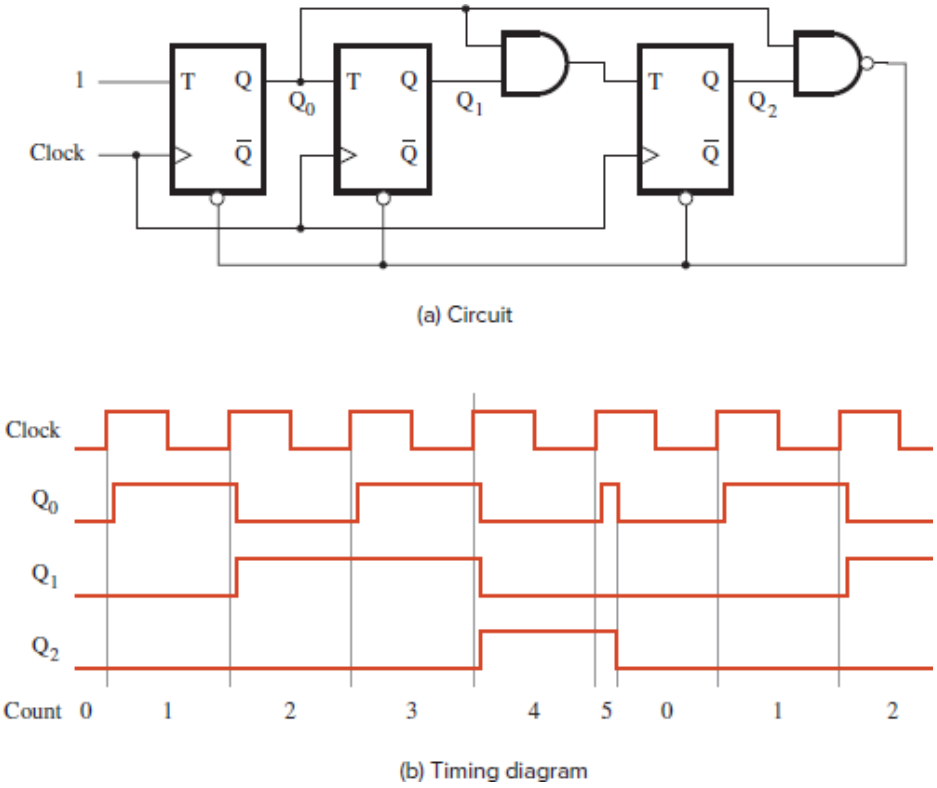


Figure 5.26 A modulo-6 counter with asynchronous reset.

The timing diagrams in Figures 5.25*b* and 5.26*b* suggest that synchronous reset is a better choice than asynchronous reset. The same observation is true if the natural counting sequence has to be broken by loading some value other than zero. The new value of the count can be established cleanly using the parallel-load feature. The alternative of using the clear and preset capability of individual flip-flops to set their states to reflect the desired count has the same problems as discussed in conjunction with the asynchronous reset.

5.11 Other Types of Counters

In this section we discuss three other types of counters that can be found in practical applications. The first uses the decimal counting sequence, and the other two generate sequences of codes that do not represent binary numbers.

5.11.1 BCD Counter

Binary-coded-decimal (BCD) counters can be designed using the approach explained in Section 5.10. A two-digit BCD counter is presented in Figure 5.27. It consists of two modulo-10 counters, one for each BCD digit, which we implemented using the parallel-load four-bit counter of Figure 5.24. Note that in a modulo-10 counter it is necessary to reset the four flip-flops after the count of 9 has been obtained. Thus the *Load* input to each stage is equal to 1 when Q_3 = Q_0 = 1, which causes 0s to be loaded into the flip-flops at the next positive edge of the clock signal. Whenever the count in stage 0, BCD_0, reaches 9 it is necessary to enable the second stage so that it will be incremented when the next clock pulse arrives. This is accomplished by keeping the *Enable* signal for BCD_1 low at all times except when BCD_0 = 9.

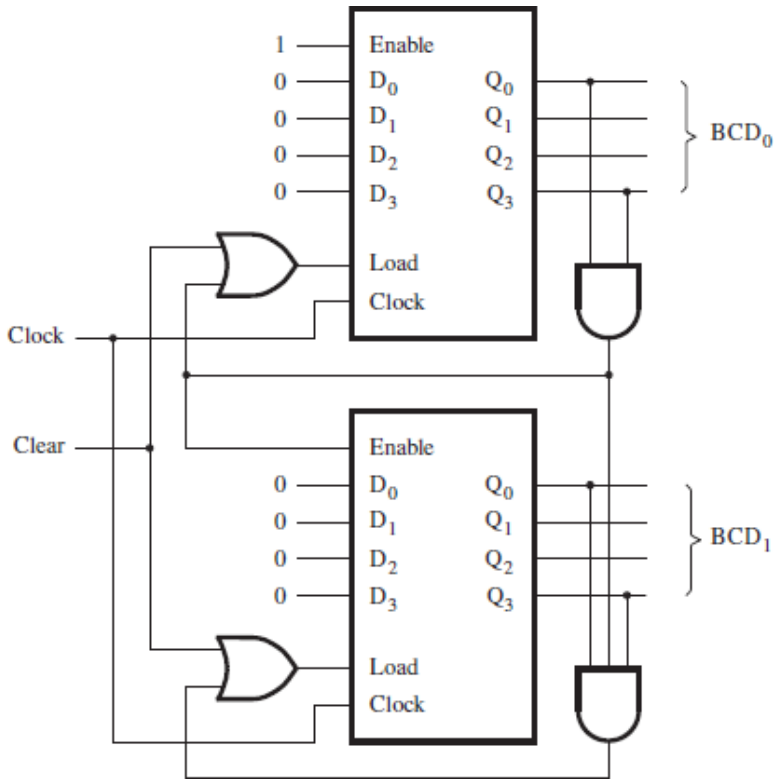


Figure 5.27 A two-digit BCD counter.

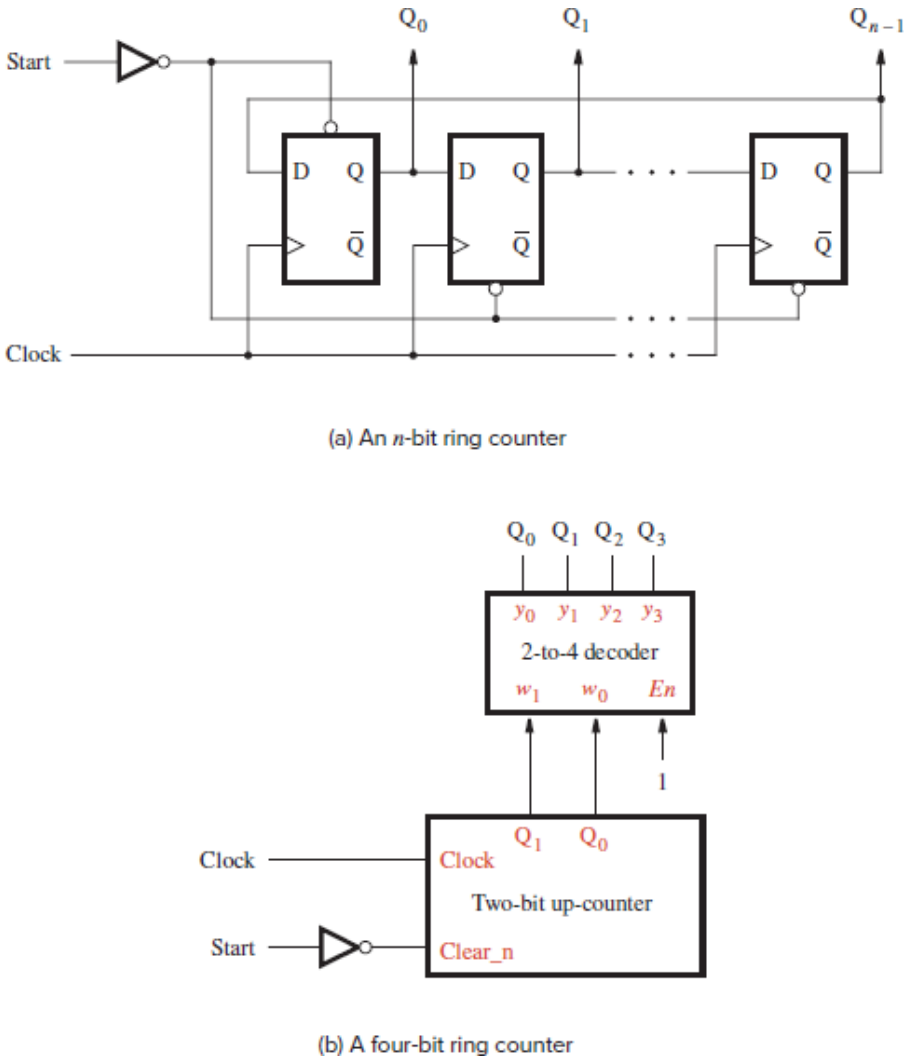
In practice, it has to be possible to clear the contents of the counter by activating some control signal. Two OR gates are included in the circuit for this purpose. The control input *Clear* can be used to load 0s into the counter. Observe that in this case *Clear* is active when high.

5.11.2 RING COUNTER

In the preceding counters the count is indicated by the state of the flip-flops in the counter. In all cases the count is a binary number. Using such counters, if an action is to be taken as a result of a particular count, then it is necessary to detect the occurrence of this count. This may be done using AND gates, as illustrated in Figures 5.25 through 5.27.

It is possible to devise a counterlike circuit in which each flip-flop reaches the state $Q_i = 1$ for exactly one count, while for all other counts $Q_i = 0$. Then Q_i indicates directly an occurrence of the corresponding count. Actually, since this does not represent binary numbers, it is better to say that the outputs of the flips-flops represent a code. Such a circuit can be constructed from a simple shift register, as indicated in Figure 5.28*a*. The Q output of the last stage in the shift register is fed back as the input to the first stage, which creates a ringlike structure. If a single 1 is injected into the ring, this 1 will be shifted through the

ring at successive clock cycles. For example, in a four-bit structure, the possible codes $Q_0Q_1Q_2Q_3$ will be 1000, 0100, 0010, and 0001. As we said in Section 4.2, such encoding, where there is a single 1 and the rest of the code variables are 0, is called a *one-hot code*.



(a) An n-bit ring counter

(b) A four-bit ring counter

Figure 5.28 Ring counter.

The circuit in Figure 5.28*a* is referred to as a *ring counter*. Its operation has to be initialized by injecting a 1 into the first stage. This is achieved by using the *Start* control signal, which presets the leftmost flip-flop to 1 and clears the others to 0. We assume that all changes in the value of the *Start* signal occur shortly after an active clock edge so that the flip-flop timing parameters are not violated.

The circuit in Figure 5.28*a* can be used to build a ring counter with any number of bits, n. For the specific case of n=4, part (*b*) of the figure shows how a ring counter can be constructed using a two-bit up-counter and a decoder. When *Start*

is set to 1, the counter is reset to 00. After *Start* changes back to 0, the counter increments its value in the normal way. The 2-to-4 decoder, described in Section 4.2, changes the counter output into a one-hot code. For the count values 00, 01, 10, 11, 00, and so on, the decoder produces $Q_0Q_1Q_2Q_3$ =1000, 0100, 0010, 0001, 1000, and so on. This circuit structure can be used for larger ring counters, as long as the number of bits is a power of two.

5.11.3 JOHNSON COUNTER

An interesting variation of the ring counter is obtained if, instead of the Q output, we take the $\overline{Q}$ output of the last stage and feed it back to the first stage, as shown in Figure 5.29. This circuit is known as a *Johnson counter*. An *n*-bit counter of this type generates a counting sequence of length 2*n*. For example, a four-bit counter produces the sequence 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000, and so on. Note that in this sequence, only a single bit has a different value for two consecutive codes.

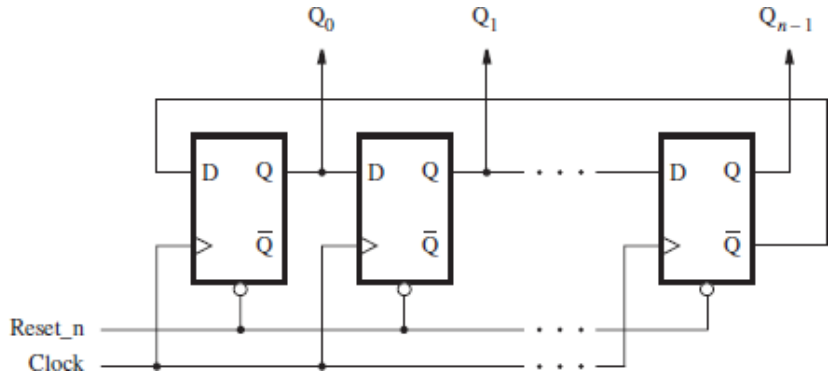


Figure 5.29 Johnson counter.

To initialize the operation of the Johnson counter, it is necessary to reset all flip-flops, as shown in the figure. Observe that neither the Johnson nor the ring counter will generate the desired counting sequence if not initialized properly.

5.11.4 REMARKS ON COUNTER DESIGN

The sequential circuits presented in this chapter, namely, registers and counters, have a regular structure that allows the circuits to be designed using an intuitive approach. In Chapter 6 we will present a more formal approach to design of sequential circuits and show how the circuits presented in this chapter can be derived using this approach.

5.12 USING STORAGE ELEMENTS WITH CAD TOOLS

This section shows how circuits with storage elements can be designed using either schematic capture or VHDL code.

5.12.1 INCLUDING STORAGE ELEMENTS IN SCHEMATICS

One way to create a circuit is to draw a schematic that builds latches and flip-flops from logic gates. Because these storage elements are used in many applications, most CAD systems provide them as prebuilt modules. Figure 5.30 shows a schematic created with a schematic capture tool, which includes three types of storage elements that are imported from a library provided as part of the CAD system. The top element is a gated D latch, the middle element is a positive-edge-triggered D flip-flop, and the bottom one is a positive-edge-triggered T flip-flop. The D and T flip-flops have asynchronous, active-low clear and preset inputs. If these inputs are not connected in a schematic, then the CAD tool makes them inactive by connecting them to the default value of 1.

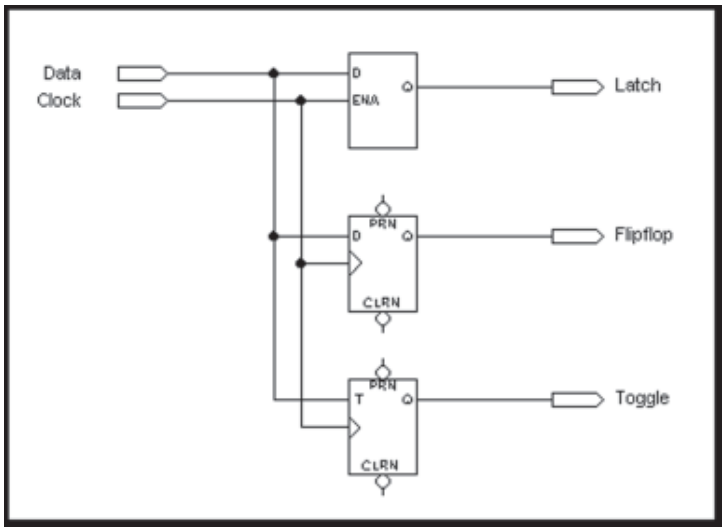


Figure 5.30 Three types of storage elements in a schematic.

When the gated D latch is synthesized for implementation in a chip, the CAD tool may not generate the cross-coupled NOR or NAND gates shown in Section 5.2. In some chips the AND-OR circuit depicted in Figure 5.31 may be preferable. This circuit is functionally equivalent to the cross-coupled version in Section 5.2. One aspect of this circuit should be mentioned. From the functional point of view, it appears that the circuit can be simplified by removing the AND gate with the inputs *Data* and *Latch*. Without this gate, the top AND gate sets the

value stored in the latch when the clock is 1, and the bottom AND gate maintains the stored value when the clock is 0. But without this gate, the circuit has a timing problem known as a *static hazard*. A detailed explanation of hazards will be given in Chapter 9.

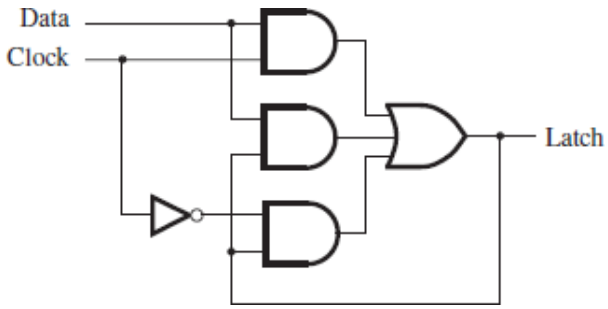


Figure 5.31 Gated D latch generated by CAD tools.

The circuit in Figure 5.30 can be implemented in a CPLD as shown in Figure 5.32. The D and T flip-flops are realized using the flip-flops on the chip that are configurable as either D or T types. The figure depicts in red the gates and wires needed to implement the circuit in Figure 5.30.

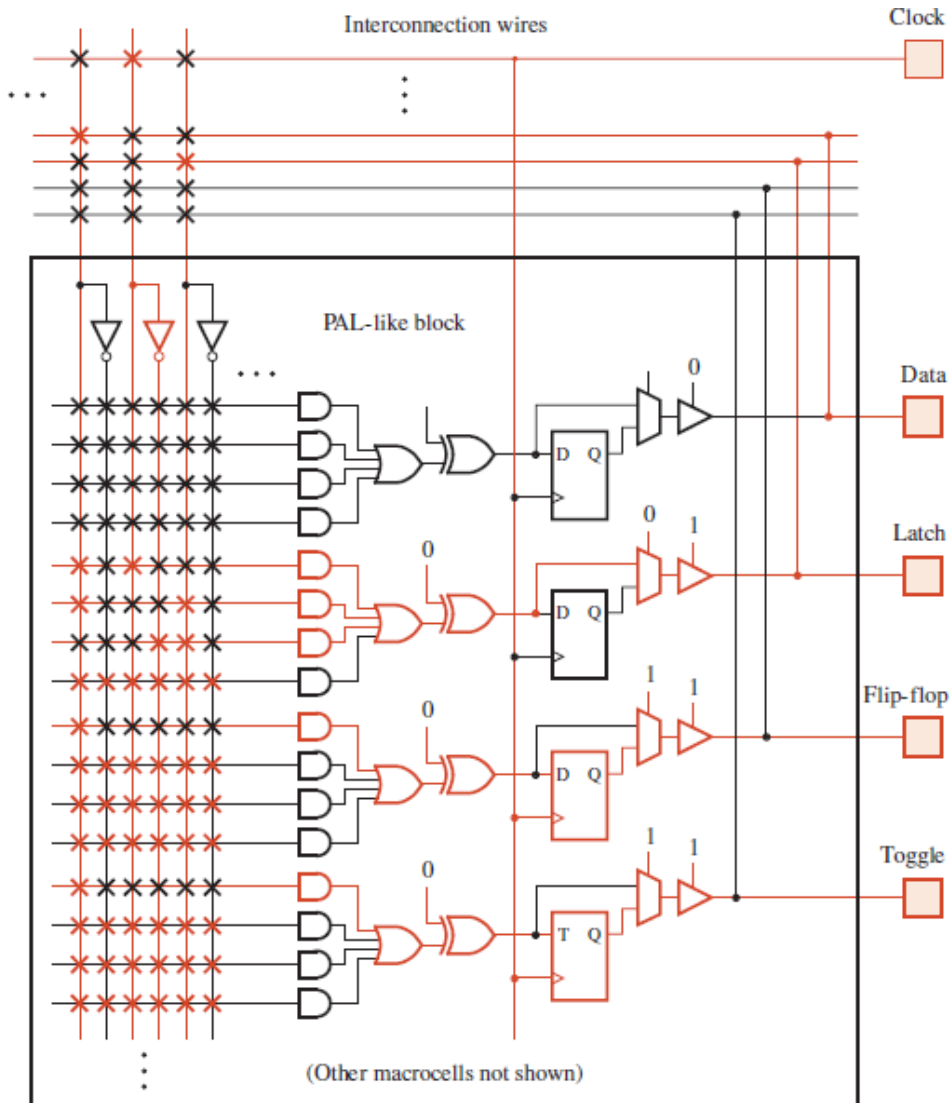


Figure 5.32 Implementation of the schematic in Figure 5.30 in a CPLD.

The results of a timing simulation for the implementation in Figure 5.32 are given in Figure 5.33. The *Latch* signal, which is the output of the gated D latch implemented as indicated in Figure 5.31, follows the *Data* input whenever the *Clock* signal is 1. Because of propagation delays in the chip, the *Latch* signal is delayed in time with respect to the *Data* signal. Since the *Flipflop* signal is the output of the D flip-flop, it changes only after a positive clock edge. Similarly, the output of the T flip-flop, called *Toggle* in the figure, toggles when *Data* = 1 and a positive clock edge occurs. The timing diagram illustrates the delay from when the positive clock edge occurs at the input pin of the chip until a change in the flip-flop output appears at the output pin of the chip. This time is called the *clock-to-output time*, t_{co}.

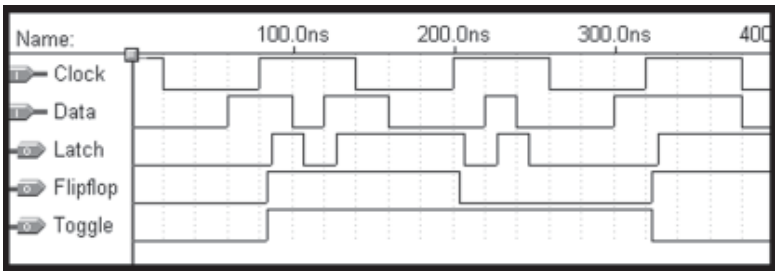


Figure 5.33 Timing simulation for the storage elements in Figure 5.30.

5.12.2 USING VHDL CONSTRUCTS FOR STORAGE ELEMENTS

In Section 4.6 we described a number of VHDL assignment statements. The IF and CASE statements were introduced as two types of sequential assignment statements. In this section we show how these statements can be used to describe storage elements.

Figure 4.39, which is repeated in Figure 5.34, gives an example of VHDL code that has implied memory. Because the code does not specify what value the *AeqB* signal should have when the condition for the IF statement is not satisfied, the semantics specify that in this case *AeqB* should retain its current value. The implied memory is the key concept used for describing sequential circuit elements, which we will illustrate using several examples.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
    PORT ( A, B  : IN   STD_LOGIC ;
           AeqB : OUT  STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.34 The code from Figure 4.39, illustrating implied memory.

Example 5.1

CODE FOR A GATED D LATCH The code in Figure 5.35 defines an entity named *latch*, which has the inputs *D* and *Clk* and the output Q. The process uses an if-then-else statement to define the value of the Q output. When *Clk* =1, Q takes the value of *D*. For the case when *Clk* is not 1, the code does not specify what value Q should have. Hence Q will retain its current value in this case, and the code describes a gated D latch. The process sensitivity list includes both *Clk* and *D* because these signals can cause a change in the value of the Q output.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
    PORT ( D, Clk  : IN    STD_LOGIC ;
           Q       : OUT   STD_LOGIC) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
    PROCESS ( D, Clk )
    BEGIN
        IF Clk = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.35 Code for a gated D latch.

Example 5.2

CODE FOR A D FLIP-FLOP Figure 5.36 defines an entity named *flipflop*, which is a positive-edge-triggered D flip-flop. The code is identical to Figure 5.35 with two exceptions. First, the process sensitivity list contains only the clock signal because it is the only signal that can cause a change in the Q output. Second, the IF statement uses a different condition from the one used in the latch. The syntax Clock'EVENT uses a VHDL construct called an *attribute*. An attribute refers to a property of an object, such as a signal. In this case the 'EVENT attribute refers to any change in the *Clock* signal. Combining the Clock'EVENT condition with the condition *Clock* = 1 means that "the value of the *Clock* signal has just changed, and the value is now equal to 1." Hence the condition refers to a positive clock edge. Because the Q output changes only as a result of a positive clock edge, the code describes a positive-edge-triggered D flip-flop.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock  : IN   STD_LOGIC ;
           Q         : OUT  STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.36 Code for a D flip-flop.

Example 5.3

ALTERNATIVE CODE FOR A D FLIP-FLOP The process in Figure 5.37 uses a different syntax from that in Figure 5.36 to describe a D flip-flop. It uses the statement WAIT UNTIL Clock'EVENT AND Clock = '1'. This statement has the same effect as the IF statement in Figure 5.36. A process that uses a WAIT UNTIL statement is a special case because the sensitivity list is omitted. The WAIT UNTIL construct implies that the sensitivity list includes only the *Clock* signal. In our use of VHDL, which is for synthesis of circuits, a process can use a WAIT UNTIL statement only if this is the first statement in the process.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
    PORT ( D, Clock  : IN   STD_LOGIC ;
           Q         : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        Q <= D ;
    END PROCESS ;
END Behavior ;
```

Figure 5.37 Equivalent code to Figure 5.36, using a WAIT UNTIL statement.

Actually, the attribute 'EVENT is redundant in the WAIT UNTIL statement. We can write simply

```
WAIT UNTIL Clock = '1' ;
```

which also implies that the action occurs when the *Clock* signal becomes equal to 1, namely, at the edge when the signal changes from 0 to 1. However, some CAD synthesis tools require the inclusion of the 'EVENT attribute, which is the reason why we use this style in the book.

In general, whenever it is desired to include in VHDL code flip-flops that are clocked by the positive clock edge, the condition Clock'EVENT AND Clock ‘1’ is used. When this condition appears in an IF statement, any signals that are assigned values inside the IF statement are implemented as the outputs of flip-flops. When the condition is used in a WAIT UNTIL statement, any signal that is assigned a value in the entire process is implemented as the output of a flip-flop.

The differences in using the IF and WAIT UNTIL statements are discussed in more detail in Appendix A, Section A.10.3.

Example 5.4

ASYNCHRONOUS CLEAR Figure 5.38 gives a process that is similar to the one in Figure 5.36. It describes a D flip-flop with an asynchronous active-low reset (clear) input. When *Resetn*, the reset input, is equal to 0, the flip-flop's Q output is set to 0.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.38 D flip-flop with asynchronous reset.

Example 5.5

SYNCHRONOUS CLEAR Figure 5.39 shows how a D flip-flop with a synchronous reset input can be described. In this case the reset signal is acted upon only when a positive clock edge arrives. The code generates the circuit in Figure 5.13*c*, which has an AND gate connected to the flip-flop's D input.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSE
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.39 D flip-flop with synchronous reset.

So far, we have specified the active edge of the clock signal as Clock'EVENT AND Clock = ‘1’, which corresponds to the time when this signal changes from 0 to 1, namely its rising edge. Instead, we can state simply RISING_EDGE (Clock). Then, a D flip-flop can be specified as shown in Figure 5.40.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Clock ) IS
    BEGIN
        IF RISING_EDGE ( Clock ) THEN
            IF Resetn = '0' THEN
                Q <= '0' ;
            ELSE
                Q <= D ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.40 An alternative specification of a D flip-flop with synchronous reset.

5.13 Using VHDL Sequential Statements for Registers and Counters

In this section we show how registers and counters can be specified in VHDL code using sequential statements. Figure 5.38 gives code for a D flip-flop. One way to describe an *n*-bit register is to write hierarchical code that includes *n* instances of the D flip-flop subcircuit. A simpler approach is shown in Figure 5.41. It uses the same code as in Figure 5.38 except that the D input and Q output are defined as multibit signals. The code represents an eight-bit register with asynchronous clear.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg8 IS
    PORT ( D              : IN   STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END reg8 ;

ARCHITECTURE Behavior OF reg8 IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= "00000000" ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.41 Code for an eight-bit register with asynchronous clear.

Example 5.6

AN *N*-BIT REGISTER Since registers of different sizes are often needed in logic circuits, it is advantageous to define a register entity for which the number of flip-flops can be easily changed. Figure 5.42 shows how the code in Figure 5.41 can be extended to include a parameter that sets the number of flip-flops. The parameter is an integer, *N*, which is defined using the VHDL construct called GENERIC. The value of *N* is set to 16 using the := assignment operator. By changing this parameter, the code can represent a register of any size. If the register is declared as a component, then it can be used as a subcircuit in other code. That code can either use the default value of the GENERIC parameter or else specify a different value.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( N : INTEGER := 16 ) ;
    PORT ( D              : IN   STD_LOGIC_VECTOR(N–1 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(N–1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.42 Code for an *n*-bit register with asynchronous clear.

The *D* and Q signals in Figure 5.42 are defined in terms of *N*. The statement that resets all the bits of Q to 0 uses the odd-looking syntax Q <= (OTHERS => ‘0’). For the default value of *N*=16, this statement is equivalent to writing Q <= “0000000000000000”. The (OTHERS => ‘0’) syntax results in a ‘0’ digit being assigned to each bit of Q, regardless of how many bits Q has. It allows the code to be used for any value of *N*, rather than only for *N*=16.

Example 5.7

A FOUR-BIT SHIFT REGISTER Assume that we wish to write VHDL code that represents the four-bit shift register in Figure 5.18. One approach is to write hierarchical code that uses four subcircuits. Each subcircuit consists of a D flip-flop with a 2-to-1 multiplexer connected to the *D* input. Figure 5.43 defines the entity named *muxdff*, which represents this subcircuit. The two data inputs are named D_0 and D_1, and they are selected using the *Sel* input. The process statement specifies that on the positive clock edge if *Sel* = 0, then Q is assigned the value of D_0; otherwise, Q is assigned the value of D_1.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY muxdff IS
    PORT ( D0, D1, Sel, Clock  : IN   STD_LOGIC ;
           Q                   : OUT  STD_LOGIC ) ;
END muxdff ;

ARCHITECTURE Behavior OF muxdff IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Sel = '0' THEN
            Q <= D0 ;
        ELSE
            Q <= D1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.43 Code for a D flip-flop with a 2-to-1 multiplexer on the *D* input.

Figure 5.44 defines the four-bit shift register. The statement labeled *Stage*3 instantiates the leftmost flip-flop, which has the output Q_3, and the statement labeled *Stage*0 instantiates the rightmost flip-flop, Q_0. When $L = 1$, it is loaded in parallel from the *R* input, and when $L = 0$, shifting takes place in the left to right direction. Serial data is shifted into the most-significant bit, Q_3, from the *w* input.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shift4 IS
    PORT ( R       : IN      STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Clock   : IN      STD_LOGIC ;
           L, w    : IN      STD_LOGIC ;
           Q       : BUFFER  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END shift4 ;

ARCHITECTURE Structure OF shift4 IS
    COMPONENT muxdff
        PORT ( D0, D1, Sel, Clock  : IN   STD_LOGIC ;
               Q                   : OUT  STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    Stage3: muxdff PORT MAP ( w, R(3), L, Clock, Q(3) ) ;
    Stage2: muxdff PORT MAP ( Q(3), R(2), L, Clock, Q(2) ) ;
    Stage1: muxdff PORT MAP ( Q(2), R(1), L, Clock, Q(1) ) ;
    Stage0: muxdff PORT MAP ( Q(1), R(0), L, Clock, Q(0) ) ;
END Structure ;
```

Figure 5.44 Hierarchical code for a four-bit shift register.

Example 5.8

ALTERNATIVE CODE FOR A FOUR-BIT SHIFT REGISTER A different style of code for the four-bit shift register is given in Figure 5.45. The lines of code are numbered for ease of reference. Instead of using subcircuits, the shift register is described using sequential statements. Due to the WAIT UNTIL statement in line 13, any signal that is assigned a value inside the process has to be implemented as the output of a flip-flop. Lines 14 and 15 specify the parallel loading of the shift register when $L = 1$. The ELSE clause in lines 16 to 20 specifies the shifting operation. Line 17 shifts the value of Q_1 into the flip-flop with the output Q_0. Lines 18 and 19 shift the values of Q_2 and Q_3 into the flip-flops with the outputs Q_1 and Q_2, respectively. Finally, line 20 shifts the value of w into the leftmost flip-flop, which has the output Q_3. Note that the process semantics, described in Section 4.6.6, stipulate that the four assignments in lines 17 to 20 are scheduled to occur only after all of the statements in the process have been evaluated. Hence all four flip-flops change their values at the same time, as required in the shift register. The code generates the same shift-register circuit as the code in Figure 5.44.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( R       : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
5              Clock   : IN     STD_LOGIC ;
6              L, w    : IN     STD_LOGIC ;
7              Q       : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
8   END shift4 ;

9   ARCHITECTURE Behavior OF shift4 IS
10  BEGIN
11      PROCESS
12      BEGIN
13          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
14          IF L = '1' THEN
15              Q <= R ;
16          ELSE
17              Q(0) <= Q(1) ;
18              Q(1) <= Q(2);
19              Q(2) <= Q(3) ;
20              Q(3) <= w ;
21          END IF ;
22      END PROCESS ;
23  END Behavior ;
```

Figure 5.45 Alternative code for a shift register.

It is instructive to consider the effect of reversing the ordering of lines 17 through 20 in Figure 5.45, as indicated in Figure 5.46. In this case the first shift operation specified in the code, in line 17, shifts the value of *w* into the leftmost flip-flop with the output Q_3. Due to the semantics of the process statement, the assignment to Q_3 does not take effect until all of the subsequent statements inside the process are evaluated. Hence line 18 shifts the present value of Q_3, before it is changed as a result of line 17, into the flip-flop with the output Q_2. Similarly, lines 19 and 20 shift the present values of Q_2 and Q_1 into the flip-flops with the outputs Q_1 and Q_0, respectively. The code produces the same circuit as it did with the ordering of the statements in Figure 5.45.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( R       : IN      STD_LOGIC_VECTOR(3 DOWNTO 0) ;
5              Clock   : IN      STD_LOGIC ;
6              L, w    : IN      STD_LOGIC ;
7              Q       : BUFFER  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
8   END shift4 ;

9   ARCHITECTURE Behavior OF shift4 IS
10  BEGIN
11      PROCESS
12      BEGIN
13          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
14          IF L = '1' THEN
15              Q <= R ;
16          ELSE
17              Q(3) <= w ;
18              Q(2) <= Q(3) ;
19              Q(1) <= Q(2);
20              Q(0) <= Q(1) ;
21          END IF ;
22      END PROCESS ;
23  END Behavior ;
```

Figure 5.46 Code that reverses the ordering of statements in Figure 5.45.

Example 5.9

***N*-BIT SHIFT REGISTER** Figure 5.47 shows code that can be used to represent shift registers of any size. The GENERIC parameter *N*, which has the default value 8

in the figure, sets the number of flip-flops. The code is identical to that in Figure 5.45 with two exceptions. First, *R* and Q are defined in terms of *N*. Second, the ELSE clause that describes the shifting operation is generalized to work for any number of flip-flops.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shiftn IS
4       GENERIC ( N : INTEGER := 8 ) ;
5       PORT ( R       : IN     STD_LOGIC_VECTOR(N  1 DOWNTO 0) ;
6              Clock   : IN     STD_LOGIC ;
7              L, w    : IN     STD_LOGIC ;
8              Q       : BUFFER STD_LOGIC_VECTOR(N  1 DOWNTO 0) ) ;
9   END shiftn ;

10  ARCHITECTURE Behavior OF shiftn IS
11  BEGIN
12      PROCESS
13      BEGIN
14          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
15          IF L = '1' THEN
16              Q <= R ;
17          ELSE
18              Genbits: FOR i IN 0 TO N-2 LOOP
19                  Q(i) <= Q(i + 1) ;
20              END LOOP ;
21              Q(N-1) <= w ;
22          END IF ;
23      END PROCESS ;
24  END Behavior ;
```

Figure 5.47 Code for an *n*-bit left-to-right shift register.

Lines 18 to 20 specify the shifting operation for the right-most N - 1 flip-flops, which have the outputs Q_{N-2} to Q_0. The construct used is called a FOR LOOP. It is similar to the FOR GENERATE statement, introduced in Section 4.6.4, which is used to generate a set of concurrent statements. The FOR LOOP is used to generate a set of sequential statements. The first loop iteration shifts the present value of Q_1 into the flip-flop with the output Q_0. The next loop iteration shifts Q_2 into the flip-flop with the output Q_1, and so on, with the final iteration shifting Q_{N-1} into the flip-flop with the output Q_{N-2}. Line 21 completes the shift operation by shifting the value of the serial input *w* into the leftmost flip-flop with the output Q_{N-1}.

Example 5.10

UP-COUNTER Figure 5.48 shows the code for a four-bit up-counter that has a reset input, *Resetn*, and an enable input, *E*. In the architecture body the flip-flops in the counter are represented by the signal named *Count*. The process statement specifies an asynchronous reset of *Count* if *Resetn* = 0. The ELSIF clause specifies that on the positive clock edge, if *E*=1, the count is incremented. If *E*=0, the code explicitly assigns *Count* <= *Count*. This statement is not required to correctly describe the counter, because of the implied memory semantics, but it may be included for clarity. The Q outputs are assigned the value of *Count* at the end of the code. The code produces the circuit shown in Figure 5.22 if the VHDL compiler opts to use T flip-flops, and it generates the circuit in Figure 5.23 (with the reset input added) if the compiler chooses D flip-flops.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY upcount IS
    PORT ( Clock, Resetn, E  : IN     STD_LOGIC ;
           Q                 : OUT  STD_LOGIC_VECTOR (3 DOWNTO 0)) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
    SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Count <= "0000" ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF E = '1' THEN
                Count <= Count + 1 ;
            ELSE
                Count <= Count ;
            END IF ;
        END IF ;
    END PROCESS ;
    Q <= Count ;
END Behavior ;
```

Figure 5.48 Code for a four-bit up-counter.

Example 5.11

USING INTEGER SIGNALS IN A COUNTER Counters are often defined in VHDL using the INTEGER type, which was introduced in Section 3.5.4. The code in Figure 5.49 defines an up-counter that has a parallel-load input in addition to a reset

input. The parallel data, *R*, as well as the counter's output, Q, are defined using the INTEGER type. Since they have the range from 0 to 15, both of these signals represent four-bit quantities. In Figure 5.48 the signal *Count* is defined to represent the flip-flops in the counter. This signal is not needed if the Q outputs have the BUFFER mode, as shown in Figure 5.49. The if-then-else statement at the beginning of the process includes the asynchronous reset. The ELSIF clause specifies that on the positive clock edge, if $L = 1$, the flip-flops in the counter are loaded in parallel from the *R* inputs. If $L = 0$, the count is incremented.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY upcount IS
    PORT ( R              : IN     INTEGER RANGE 0 TO 15 ;
           Clock, Resetn, L : IN     STD_LOGIC ;
           Q              : BUFFER INTEGER RANGE 0 TO 15 ) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Q <= 0 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF L = '1' THEN
                Q <= R ;
            ELSE
                Q <= Q + 1 ;
            END IF;
        END IF;
    END PROCESS;
END Behavior;
```

Figure 5.49 A four-bit counter with parallel load, using INTEGER signals.

Example 5.12

DOWN-COUNTER Figure 5.50 shows the code for a down-counter named *downcnt*. To make it easy to change the starting count, it is defined as a GENERIC parameter named *modulus*. On the positive clock edge, if L=1, the counter is loaded with the value *modulus*-1, and if L=0, the count is decremented. The counter also includes an enable input, *E*. Setting E=1 allows the count to be decremented when an active clock edge occurs.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY downcnt IS
    GENERIC ( modulus : INTEGER := 8 ) ;
    PORT ( Clock, L, E  : IN     STD_LOGIC ;
           Q            : OUT  INTEGER RANGE 0 TO modulus–1 ) ;
END downcnt ;

ARCHITECTURE Behavior OF downcnt IS
    SIGNAL Count : INTEGER RANGE 0 TO modulus–1 ;
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL (Clock'EVENT AND Clock = '1') ;
        IF L = '1' THEN
            Count <= modulus–1 ;
        ELSE
            IF E = '1' THEN
                Count <= Count–1 ;
            END IF ;
        END IF ;
    END PROCESS;
    Q <= Count ;
END Behavior ;
```

Figure 5.50 Code for a down-counter.

5.13.1 FLIP-FLOPS AND REGISTERS WITH ENABLE INPUTS

We showed in Figures 5.22 and 5.23 how an enable input can be used in counter circuits to be able to prevent the flip-flops from toggling when an active clock edge occurs. It is also useful in many other types of circuits to be able to prevent the data stored in flip-flops from changing when an active clock edge occurs. For D flip-flops, this capability can be provided by adding a multiplexer to the flip-flop, as shown in Figure 5.51*a*. When E=0, the flip-flop output cannot change, because the multiplexer connects Q to the input of the flip-flop. But if E=1, then the multiplexer allows new data to be loaded into the flip-flop from the D input. Instead of using the multiplexer shown in the figure, another way to implement the enable feature is to use a two-input AND gate as illustrated in Figure 5.51*b*. Then setting E=0 prevents the clock signal from reaching the flip-flop's clock input. This method seems simpler than the multiplexer approach, but we will show in Section 5.15 that it can cause problems in practical operation. We will prefer the multiplexer-based approach over gating the clock with an AND gate.

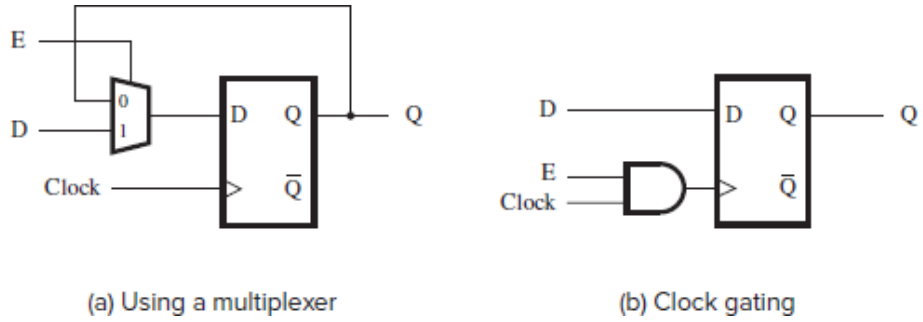


Figure 5.51 Providing an enable input for a D flip-flop.

VHDL code for a D flip-flop with an asynchronous reset input and an enable input is given in Figure 5.52. After first specifying the reset condition, the PROCESS uses an IF-THEN-ELSE statement to ensure that the data stored in the flip-flop can change only when $E = 1$. We can extend the enable capability to registers with n bits by using n 2-to-1 multiplexers controlled by E, as shown in Figure 5.53. The multiplexer for each flip-flop, i, selects either the external data bit, R_i, or the flip-flop's output, Q_i.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY rege IS
    PORT ( R, Resetn, E, Clock  : IN      STD_LOGIC ;
           Q                    : BUFFER  STD_LOGIC ) ;
END rege ;

ARCHITECTURE Behavior OF rege IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            IF E = '1' THEN
                Q <= R ;
            ELSE
                Q <= Q ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.52 Code for a D flip-flop with enable.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regne IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT ( R          : IN   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           Resetn     : IN   STD_LOGIC ;
           E, Clock   : IN   STD_LOGIC ;
           Q          : OUT  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regne ;

ARCHITECTURE Behavior OF regne IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0' ) ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            IF E = '1' THEN
                Q <= R ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.53 An *n*-bit register with an enable input.

5.13.2 SHIFT REGISTERS WITH ENABLE INPUTS

It is useful to be able to inhibit the shifting operation in a shift register by using an enable input, *E*. We showed in Figure 5.18 that shift registers can be constructed with a parallel-load capability, which is implemented using a multiplexer. Figure 5.54 shows how the enable feature can be added by using an additional multiplexer. If the parallel-load control input, *L*, is 1, the flip-flops are loaded in parallel. If *L*=0, the additional multiplexer selects new data to be loaded into the flip-flops only if the enable *E* is 1.

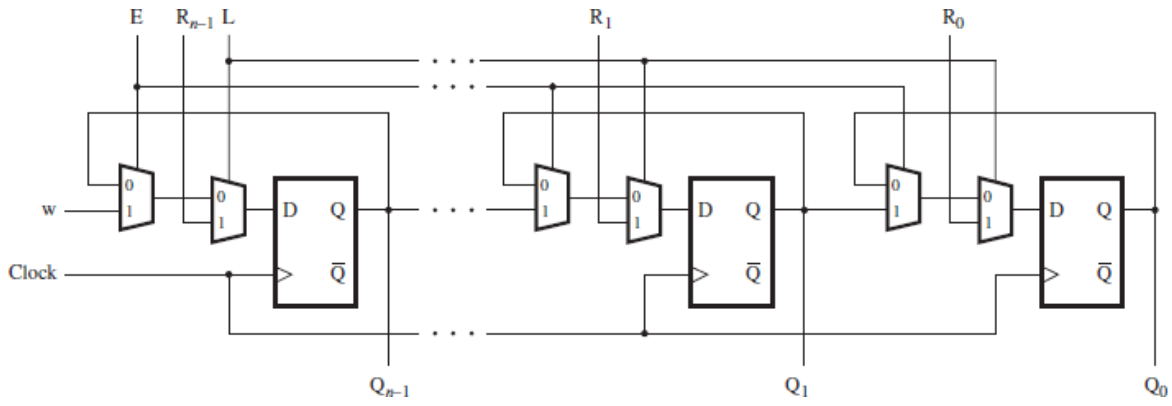


Figure 5.54 A shift register with parallel-load and enable control inputs.

VHDL code that represents the circuit in Figure 5.54 is given in Figure 5.55. When L=1, the register is loaded in parallel from the R input. When L=0 and E=1, the data in the shift register is shifted in a left-to-right direction.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

-- left-to-right shift register with parallel load and enable
ENTITY shiftrne IS
    GENERIC ( N : INTEGER := 4 ) ;
    PORT( R       : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
          L, E, w : IN     STD_LOGIC ;
          Clock   : IN     STD_LOGIC ;
          Q       : BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END shiftrne ;

ARCHITECTURE Behavior OF shiftrne IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF L = '1' THEN
            Q <= R ;
        ELSIF E = '1' THEN
            Q(N-1) <= w ;
            Genbits: FOR i IN N-2 DOWNTO 0 LOOP
                Q(i) <= Q(i+1) ;
            END LOOP ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 5.55 A left-to-right shift register with an enable input.

5.14 Design Example - Reaction Timer

This section presents an example of a digital system that makes use of some of the building blocks described in this chapter and in Chapter 4.

Electronic devices operate at remarkably fast speeds, with the typical delay through a logic gate being less than 1 ns. In this example we use a logic circuit to measure the speed of a much slower type of device—a person. We will design a circuit that can be used to measure the reaction time of a person to a specific event. The circuit turns on a small light, called a *light-emitting diode (LED)*. In response to the LED being turned on, the person attempts to press a switch as quickly as possible. The circuit measures the elapsed time from when the LED is turned on until the switch is pressed.

To measure the reaction time, a clock signal with an appropriate frequency is needed. In this example we use a 100 Hz clock, which measures time at a resolution of 1/100 of a second. The reaction time can then be displayed using two digits that represent fractions of a second from 00/100 to 99/100.

Digital systems often include high-frequency clock signals to control various subsystems. In this case assume the existence of an input clock signal with the frequency 102.4 kHz. From this signal we can derive the required 100 Hz signal by using a counter as a *clock divider*. A timing diagram for a four-bit counter is given in Figure 5.21. It shows that the least-significant bit output, Q_0, of the counter is a periodic signal with half the frequency of the clock input. Hence we can view Q_0 as dividing the clock frequency by two. Similarly, the Q_1 output divides the clock frequency by four. In general, output Q_i in an *n*-bit counter divides the clock frequency by 2^{i+1}. In the case of our 102.4 kHz clock signal, we can use a 10-bit counter, as shown in Figure 5.56*a*. The counter output c_9 has the required 100 Hz frequency because 102400 Hz/1024 = 100 Hz.

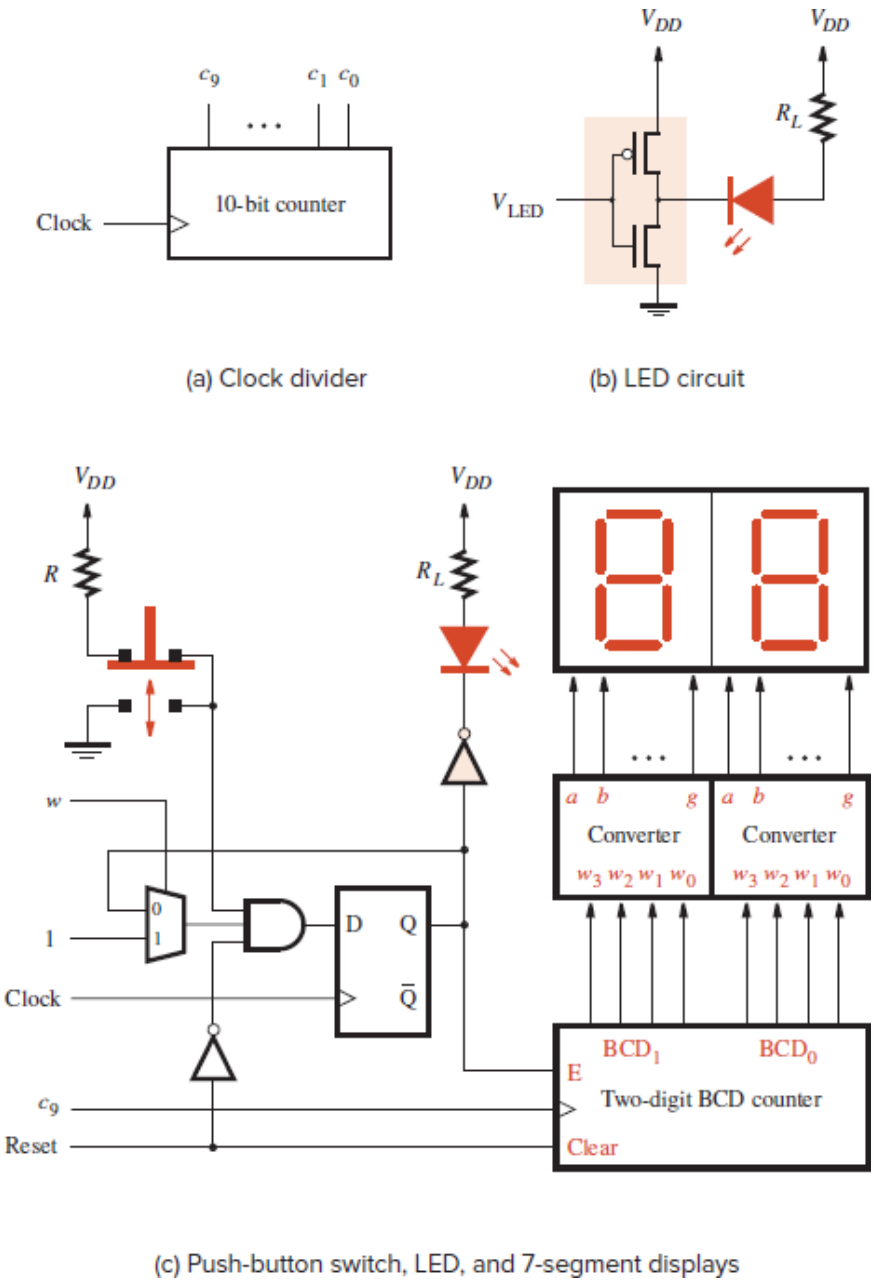


Figure 5.56 A reaction-timer circuit.

The reaction timer circuit has to be able to turn an LED on and off. The graphical symbol for an LED is shown in red in Figure 5.56*b*. Small red arrows in the symbol represent the light that is emitted when the LED is turned on. The LED has two terminals: the one on the left in the figure is the *cathode*, and the terminal on the right is the *anode*. To turn the LED on, the cathode has to be set to a sufficiently lower voltage than the anode, which causes a current to flow through the LED. If the voltages on its two terminals are equal, the LED is off.

Figure 5.56*b* shows one way to control the LED. The two transistors in this diagram implement an inverter as explained in Appendix B. If the input voltage to this inverter $V_{LED} = 0$, then the voltage at the cathode is equal to V_{DD}; hence the LED is off. But if $V_{LED} = V_{DD}$, the cathode voltage is 0 V and the LED is on. The amount of current that flows is limited by the value of the resistor R_L. This current flows through the LED and the NMOS transistor in the inverter. Since the current flows *into* the inverter, we say that the inverter *sinks* the current. The maximum current that a logic gate can sink without sustaining permanent damage is usually called I_{OL}, which stands for the "maximum current when the output is low." The value of R_L is chosen such that the current is less than I_{OL}. As an example assume that the inverter is implemented inside a chip for which the value of I_{OL} specified in the data sheet for the chip is 12 mA. For $V_{DD} = 5$ V, this leads to $R_L \approx 450\ \Omega$ because $5\ V/450\ \Omega = 11$ mA (there is actually a small voltage drop across the LED when it is turned on, but we ignore this for simplicity). The amount of light emitted by the LED is proportional to the current flow. If 11 mA is insufficient, then the inverter should be implemented in a driver chip, like those described in Appendix B, because drivers provide a higher value of I_{OL}.

The complete reaction-timer circuit is illustrated in Figure 5.56*c*, with the inverter from part (*b*) shaded in a red color. The graphical symbol for a push-button switch is shown in the top left of the diagram. The switch normally makes contact with the top terminals, as depicted in the figure. When depressed, the switch makes contact with the bottom terminals; when released, it automatically springs back to the top position. In the figure the switch is connected such that it normally produces a logic value of 1, and it produces a 0 pulse when pressed.

When depressed, the push-button switch causes the D flip-flop to be synchronously reset. The output of this flip-flop determines whether the LED is on or off, and it also provides the count enable input to a two-digit BCD counter. As discussed in Section 5.11, each digit in a BCD counter has four bits that take the values 0000 to 1001. Thus the counting sequence can be viewed as decimal numbers from 00 to 99. A circuit for the BCD counter is given in Figure 5.27. Each digit in the counter is connected through a code converter to a 7-segment display. The counter is clocked by the c_9 output of the clock divider in part (*a*) of the figure. The intended use of the reaction-timer circuit is to first assert the *Reset* input to clear the flip-flop and thus turn off the LED and disable the counter. Asserting the *Reset* input also clears the contents of the counter to 00. The input *w* normally has the value 0, which keeps the flip-flop cleared and prevents the count value from changing. The reaction test is initiated by setting w=1 for one c_9 clock cycle. After the next positive edge of the clock, the flip-flop output becomes a 1,

which turns on the LED. We assume that *w* returns to 0 after one c_9 clock cycle, but the flip-flop output remains at 1 because of the 2-to-1 multiplexer connected to the D input. The counter is then incremented every 1/100 of a second. When the user depresses the switch, the flip-flop is cleared, which turns off the LED and stops the counter. The two-digit display shows the elapsed time to the nearest 1/100 of a second from when the LED was turned on until the user was able to respond by depressing the switch.

VHDL Code

To describe the circuit in Figure 5.56*c* using VHDL code, we can make use of subcircuits for the BCD counter and the 7-segment code converter. Code for the BCD counter, which represents the circuit in Figure 5.27, is shown in Figure 5.57. The two-digit BCD output is represented by the 2 four-bit signals *BCD*1 and *BCD*0. The *Clear* input is used to provide a synchronous reset for both digits in the counter. If *E*=1, the count value is incremented on the positive clock edge, and if *E*=0, the count value is unchanged. Each digit can take the values from 0000 to 1001. Figure 5.58 gives the code for the BCD-to-7-segment decoder.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY BCDcount IS
    PORT ( CountClock     : IN      STD_LOGIC ;
           Clear, E       : IN      STD_LOGIC ;
           BCD1, BCD0     : BUFFER  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END BCDcount ;

ARCHITECTURE Behavior OF BCDcount IS
BEGIN
    PROCESS ( CountClock )
    BEGIN
        IF RISING_EDGE (CountClock) THEN
            IF Clear = '1' THEN
                BCD1 <= "0000" ; BCD0 <= "0000" ;
            ELSIF E = '1' THEN
                IF BCD0 = "1001" THEN
                    BCD0 <= "0000" ;
                    IF BCD1 = "1001" THEN
                        BCD1 <= "0000" ;
                    ELSE
                        BCD1 <= BCD1 + '1' ;
                    END IF ;
                ELSE
                    BCD0 <= BCD0 + '1' ;
                END IF ;
            END IF ;
        END IF;
    END PROCESS;
END Behavior ;
```

Figure 5.57 Code for the two-digit BCD counter in Figure 5.27.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY seg7 IS
    PORT ( bcd   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           leds  : OUT   STD_LOGIC_VECTOR(1 TO 7) ) ;
END seg7 ;

ARCHITECTURE Behavior OF seg7 IS
BEGIN
    PROCESS ( bcd )
    BEGIN
        CASE bcd IS                    --   abcdefg
            WHEN "0000"    => leds <= "1111110" ;
            WHEN "0001"    => leds <= "0110000" ;
            WHEN "0010"    => leds <= "1101101" ;
            WHEN "0011"    => leds <= "1111001" ;
            WHEN "0100"    => leds <= "0110011" ;
            WHEN "0101"    => leds <= "1011011" ;
            WHEN "0110"    => leds <= "1011111" ;
            WHEN "0111"    => leds <= "1110000" ;
            WHEN "1000"    => leds <= "1111111" ;
            WHEN "1001"    => leds <= "1111011" ;
            WHEN OTHERS => leds <= "-------" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 5.58 Code for the BCD-to-7-segment decoder.

Figure 5.59 gives the code for the reaction timer. The input signal *Pushn* represents the value produced by the push-button switch. The output signal *LEDn* represents the output of the inverter that is used to control the LED. The two 7-segment displays are controlled by the seven-bit signals *Digit* 1 and *Digit* 0.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reaction IS
    PORT ( Clock, c9, Reset   : IN      STD_LOGIC ;
           w, Pushn           : IN      STD_LOGIC ;
           LEDn               : OUT     STD_LOGIC ;
           Digit1, Digit0     : BUFFER  STD_LOGIC_VECTOR(1 TO 7) ) ;
END reaction ;

ARCHITECTURE Behavior OF reaction IS
    COMPONENT BCDcount
        PORT ( CountClock     : IN      STD_LOGIC ;
               Clear, E       : IN      STD_LOGIC ;
               BCD1, BCD0     : BUFFER  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
    END COMPONENT ;
    COMPONENT seg7
        PORT ( bcd    : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
               leds   : OUT   STD_LOGIC_VECTOR(1 TO 7) ) ;
    END COMPONENT ;
    SIGNAL LED : STD_LOGIC ;
    SIGNAL BCD1, BCD0 : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
BEGIN
    flipflop: PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Pushn = '0' OR Reset = '1' THEN
            LED <= '0' ;
        ELSIF w = '1' THEN
            LED <= '1' ;
        END IF ;
    END PROCESS ;

    LEDn <= NOT LED ;
    counter: BCDcount PORT MAP ( c9, Reset, LED, BCD1, BCD0 ) ;
    seg1 : seg7 PORT MAP ( BCD1, Digit1 ) ;
    seg0 : seg7 PORT MAP ( BCD0, Digit0 ) ;
END Behavior ;
```

Figure 5.59 Code for the reaction timer.

The flip-flop in Figure 5.56*c* is loaded with the value 1 if w=1, but if w=0 the stored value in the flip-flop is not changed. This circuit is described by the process labeled *flipflop* in Figure 5.59, which also includes a synchronous reset input. We have chosen to use a synchronous reset because the flip-flop output is connected to the enable input E on the BCD counter. As we know from the discussion in Section 5.3, it is important that all signals connected to flip-flops meet the required setup and hold times. The push-button switch can be pressed at any time and is not synchronized to the c_9 clock signal. By using a synchronous reset for the flip-flop in Figure 5.56, we avoid possible timing problems in the counter. Of course, the setup time of the flip-flop itself may become violated due

to the asynchronous operation of the push-button switch. We show in Chapter 7 how this type of problem can be alleviated by adding extra flip-flops that are used to synchronize signals.

A simulation result for the reaction-timer circuit is shown in Figure 5.60. Initially, *Reset* is asserted to clear the flip-flop and counter. When *w* changes to 1, the circuit sets *LEDn* to 0, which represents the LED being turned on. After some amount of time, the switch will be depressed. In the simulation we arbitrarily set *Pushn* to 0 after 16 c_9 clock cycles. Thus this choice represents the case when the person's reaction time is about 0.16 seconds. In human terms this duration is a very short time; for electronic circuits it is a very long time. An inexpensive personal computer can perform tens of millions of operations in 0.16 seconds!

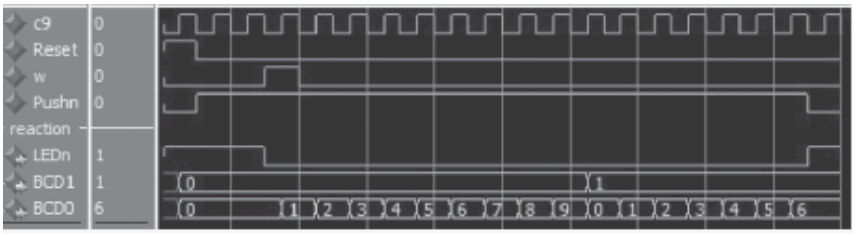


Figure 5.60 Simulation of the reaction timer circuit.

5.14.1 REGISTER TRANSFER LEVEL (RTL) CODE

At this point, we have introduced many of the VHDL constructs that are needed for synthesis. Most of our examples give behavioral code, utilizing IF-THEN-ELSE statements, CASE statements, FOR loops, and so on. It is possible to write behavioral code in a style that resembles a computer program, in which there is a complex flow of control with many loops and branches. With such code, sometimes called *high-level* behavioral code, it is difficult to relate the code to the final hardware implementation; it may even be difficult to predict what circuit a high-level synthesis tool will produce. In this book we do not use the high-level style of code. Instead, we present VHDL code in such a way that the code can be easily related to the circuit that is being described. Most design modules presented are fairly small, to facilitate simple descriptions. Larger designs are built by interconnecting the smaller modules. This approach is usually referred to as the *register-transfer level* (RTL) style of code. It is the most popular design method used in practice. RTL code is characterized by a straightforward flow of control through the code; it comprises well-understood subcircuits that are connected together in a simple way.

5.15 Timing Analysis of Flip-flop Circuits

In Figure 5.14 we showed the timing parameters associated with a D flip-flop. A simple circuit that uses this flip-flop is given in Figure 5.61. We wish to calculate the maximum clock frequency, F_{max}, for which this circuit will operate properly, and also determine if the circuit suffers from any hold time violations. In the literature, this type of analysis of circuits is usually called *timing analysis*. We will assume that the flip-flop timing parameters have the values t_{su}=0.6 ns, t_h=0.4 ns, and 0.8 ns ≤ t_{cQ} ≤ 1.0 ns. A range of minimum and maximum values is given for t_{cQ} because, as we mentioned in Section 5.4.4, this is the usual way of dealing with variations in delay that exist in integrated circuit chips.

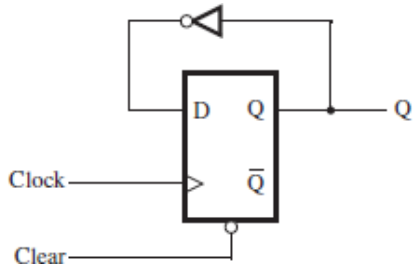


Figure 5.61 A simple flip-flop circuit.

To calculate the minimum period of the clock signal, T_{min}=1/F_{max}, we need to consider all paths in the circuit that start and end at flip-flops. In this simple circuit there is only one such path, which starts when data is loaded into the flip-flop by a positive clock edge, propagates to the Q output after the t_{cQ} delay, propagates through the NOT gate, and finally must meet the setup requirement at the D input. Therefore

$$T_{min} = t_{cQ} + t_{NOT} + t_{su}$$

Since we are interested in the longest delay for this calculation, the maximum value of t_{cQ} should be used. For the calculation of t_{NOT} we will assume that the delay through any logic gate can be calculated as 1 + 0.1k, where k is the number of inputs to the gate. For a NOT gate this gives 1.1 ns, which leads to

$$T_{min} = 1.0 + 1.1 + 0.6 = 2.7 \text{ ns}$$
$$F_{max} = 1/2.7 \text{ ns} = 370.37 \text{ MHz}$$

It is also necessary to check if there are any hold time violations in the circuit. In this case we need to examine the shortest possible delay from a positive clock

edge to a change in the value of the D input. The delay is given by $t_{cQ} + t_{NOT} = 0.8 + 1.1 = 1.9$ ns. Since 1.9 ns > $t_h = 0.4$ ns there is no hold time violation.

As another example of timing analysis of flip-flop circuits, consider the counter circuit shown in Figure 5.62. We wish to calculate the maximum clock frequency for which this circuit will operate properly assuming the same flip-flop timing parameters as we did for Figure 5.61. We will again assume that the propagation delay through a logic gate can be calculated as $1 + 0.1k$.

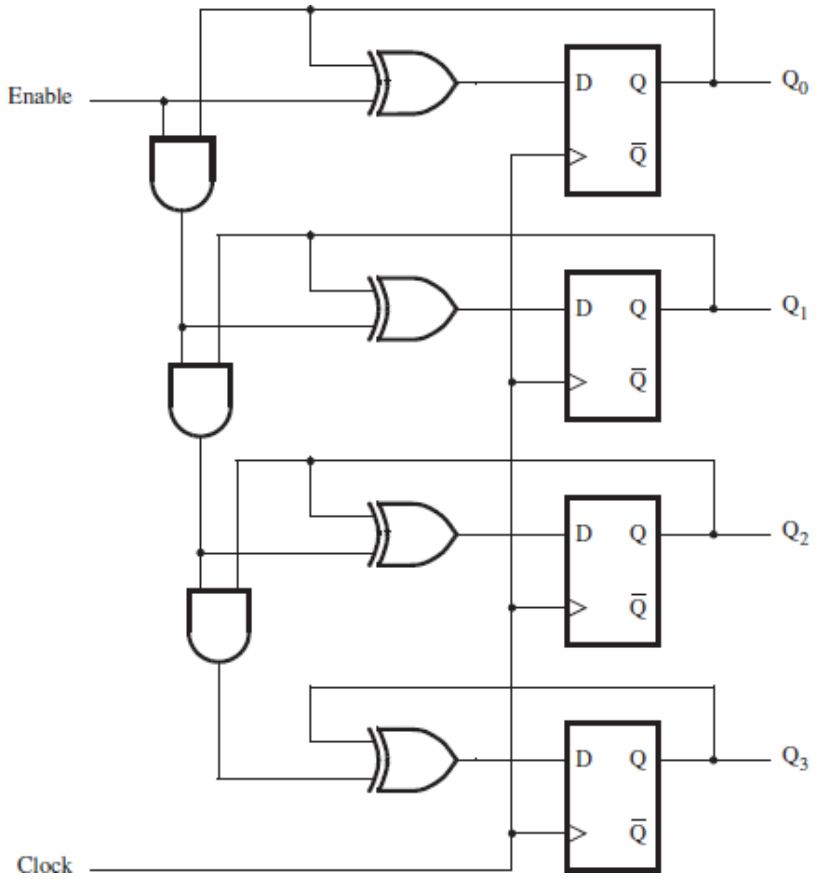


Figure 5.62 A four-bit counter.

There are many paths in this circuit that start and end at flip-flops. The longest such path starts at flip-flop Q_0 and ends at flip-flop Q_3. The longest path in a circuit is often called a *critical* path. The delay of the critical path includes the clock-to-Q delay of flip-flop Q_0, the propagation delay through three AND gates, and one XOR-gate delay. We must also account for the setup time of flip-flop Q_3. This gives

$$T_{min} = t_{cQ} + 3(t_{AND}) + t_{XOR} + t_{su}$$

Using the maximum value of t_{cQ} gives

$$T_{min} = 1.0 + 3(1.2) + 1.2 + 0.6 \text{ ns} = 6.4 \text{ ns}$$
$$F_{max} = 1/6.4 \text{ ns} = 156.25 \text{ MHz}$$

The shortest paths through the circuit are from each flip-flop to itself, through an XOR gate. The minimum delay along each such path is $t_{cQ} + t_{XOR} = 0.8 + 1.2 = 2.0$ ns. Since 2.0 ns > $t_h = 0.4$ ns, there are no hold time violations.

5.15.1 TIMING ANALYSIS WITH CLOCK SKEW

In the above analysis we assumed that the clock signal arrived at exactly the same time at all four flip-flops. We will now repeat this analysis assuming that the clock signal still arrives at flip-flops Q_0, Q_1, and Q_2 simultaneously, but that there is a delay in the arrival of the clock signal at flip-flop Q_3. Such a variation in the arrival time of a clock signal at different flip-flops is called *clock skew*, t_{skew}, and can be caused by a number of factors.

In Figure 5.62 the critical path through the circuit is from flip-flop Q_0 to Q_3. However, the clock skew at Q_3 has the effect of "reducing" this delay, because it provides additional time before data is loaded into this flip-flop. Taking a clock skew of 1.5 ns into account, the delay of the path from flip-flop Q_0 to Q_3 is given by $t_{cQ} + 3(t_{AND}) + t_{XOR} + t_{su} - t_{skew} = 6.4\text{-}1.5$ ns = 4.9 ns. There is now a different critical path through the circuit, which starts at flip-flop Q_0 and ends at Q_2. The delay of this path gives

$$\begin{aligned} T_{min} &= t_{cQ} + 2(t_{AND}) + t_{XOR} + t_{su} \\ &= 1.0 + 2(1.2) + 1.2 + 0.6 \text{ ns} \\ &= 5.2 \text{ ns} \\ F_{max} &= 1/5.2 \text{ ns} = 192.31 \text{ MHz} \end{aligned}$$

In this case the clock skew results in an increase in the circuit's maximum clock frequency. But if the clock skew had been negative, which would be the case if the clock signal arrived earlier at flip-flop Q_3 than at other flip-flops, then the result would have been a reduced F_{max}.

Since the loading of data into flip-flop Q_3 is delayed by the clock skew, it has the effect of increasing the hold time requirement of this flip-flop to $t_h + t_{skew}$, for all paths that end at Q_3 but start at Q_0, Q_1, or Q_2. The shortest such path in the

circuit is from flip-flop Q_2 to Q_3 and has the delay $t_{cQ} + t_{AND} + t_{XOR} = 0.8 + 1.2 + 1.2 = 3.2$ ns. Since 3.2 ns > $t_h + t_{skew} = 1.9$ ns there is no hold time violation.

If we repeat the above hold time analysis for clock skew values $t_{skew} \geq 3.2 - t_h = 2.8$ ns, then hold time violations will exist. Thus, if $t_{skew} \geq 2.8$ ns the circuit will not work reliably at any clock frequency.

Example 3.13

consider the circuit in Figure 5.63. In this circuit, there is a path that starts at flip-flop Q_1, passes through some network of logic gates, and ends at the D input of flip-flop Q_2. As indicated in the figure, different delays may be incurred before the clock signal reaches the flip-flops. Let Δ_1 and Δ_2 be the clock-signal delays for flip-flops Q_1 and Q_2, respectively. The clock skew between these two flip-flops is then defined as

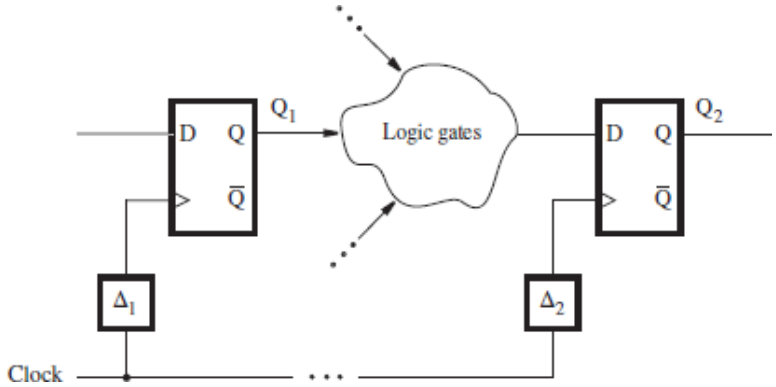


Figure 5.63 A general example of clock skew.

$$t_{skew} = \Delta_2 - \Delta_1$$

Let the longest delay along the paths through the logic gates in the circuit be t_L. Then, the minimum allowable clock period for these two flip-flops is

$$T_{min} = t_{cQ} + t_L + t_{su} - t_{skew}$$

Thus, if $\Delta_2 > \Delta_1$, then t_{skew} allows for an increased value of F_{max}, but if $\Delta_2 < \Delta_1$, then the clock skew requires a decrease in F_{max}.

To calculate whether a hold time violation exists at flip-flop Q_2 we need to determine the delay along the shortest path between the flip-flops. If the

minimum delay through the logic gates in the circuit is t_l, then a hold time violation occurs if

$$t_{cQ} + t_l < t_h + t_{skew}$$

Here, the hold-time constraint is more difficult to meet if $\Delta_2 - \Delta_1 > 0$, and it is less difficult to meet if $\Delta_2 - \Delta_1 < 0$.

The techniques described above for F_{max} and hold time analysis can be applied to any circuit in which the same, or related, clock signals are connected to all flip-flops. Consider again the reaction-timer circuit in Figure 5.56. The clock divider in part (*a*) of the figure generates the c_9 signal, which drives the clock inputs of the flip-flops in the BCD counter. Since these paths start at the Q output of the c_9 flip-flop but end at the clock inputs of other flip-flops, rather than at D inputs, the above timing analysis method cannot be applied. However, we could restructure the reaction-timer circuit as indicated in Figure 5.64. Instead of using c_9 directly as a clock for the BCD counter, a ten-input AND gate is used to generate a signal that has the value 1 for only one of the 1024 count values from the clock divider. This signal is then ANDed with the control flip-flop output to cause the BCD counter to be incremented at the desired rate of 100 times per second.

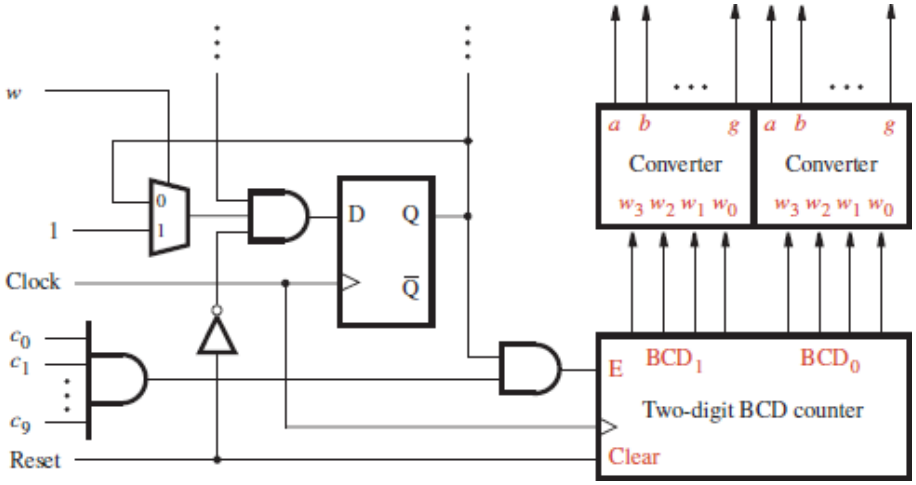


Figure 5.64 A modified version of the reaction-timer circuit.

In the circuit of Figure 5.64 all flip-flops are clocked directly by the *Clock* signal. Therefore, the F_{max} and hold time analysis can now be applied. In general, it is a good design approach for sequential circuits to connect the clock inputs of all flip-flops to a common clock. We discuss such issues in detail in Chapter 7.

5.16 Concluding Remarks

In this chapter we have presented circuits that serve as basic storage elements in digital systems. These elements are used to build larger units such as registers, shift registers, and counters. We have illustrated how circuits with flip-flops can be described using VHDL code. In the next chapter a more formal method for designing circuits with flip-flops will be presented.

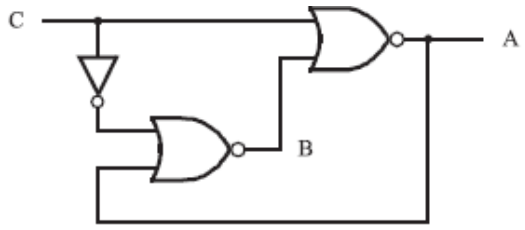
5.17 Examples of Solved Problems

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

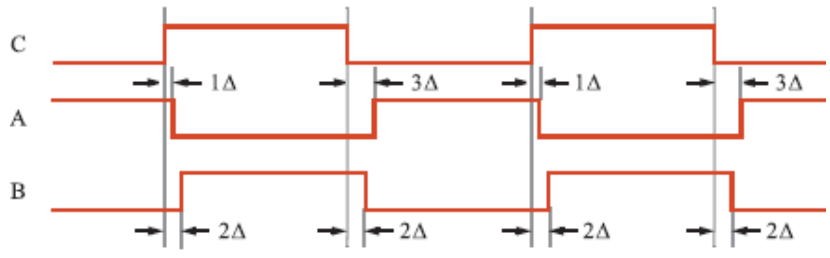
Example 5.14

Problem: Consider the circuit in Figure 5.65*a*. Assume that the input *C* is driven by a square wave signal with a 50% duty cycle. Draw a timing diagram that shows the waveforms at points *A* and *B*. Assume that the propagation delay through each gate is Δ seconds.

Solution: The timing diagram is shown in Figure 5.65*b*.



(a) Circuit



(b) Timing diagram

Figure 5.65 Circuit for Example 5.14.

Example 5.15

Problem: Determine the functional behavior of the circuit in Figure 5.66. Assume that input *w* is driven by a square wave signal.

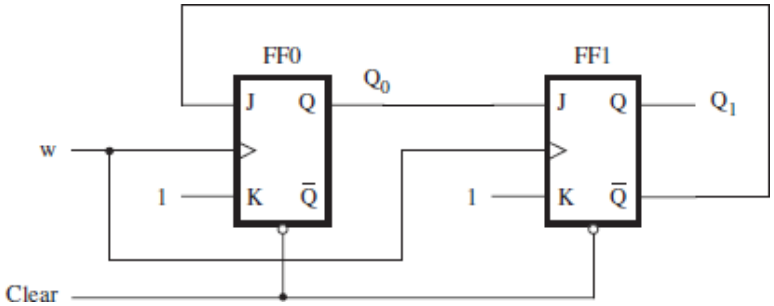


Figure 5.66 Circuit for Example 5.15.

Solution: When both flip-flops are cleared, their outputs are $Q_0 = Q_1 = 0$. After the *Clear* input goes high, each pulse on the *w* input will cause a change in the flip-flops as indicated in Figure 5.67. Note that the figure shows the state of the signals after the changes caused by the rising edge of a pulse have taken place.

Time interval	FF0			FF1		
	J_0	K_0	Q_0	J_1	K_1	Q_1
Clear	1	1	0	0	1	0
t_1	1	1	1	1	1	0
t_2	0	1	0	0	1	1
t_3	1	1	0	0	1	0
t_4	1	1	1	1	1	0

Figure 5.67 Summary of the behavior of the circuit in Figure 5.66.

In consecutive time intervals the values of Q_1Q_0 are 00, 01, 10, 00, 01, and so on. Therefore, the circuit generates the counting sequence: 0, 1, 2, 0, 1, and so on. Hence, the circuit is a modulo-3 counter.

Example 5.16

Problem: Design a circuit that can be used to control a vending machine. The circuit has five inputs: Q (quarter), D (dime), N (nickel), *Coin*, and *Resetn*. When

a coin is deposited in the machine, a coin-sensing mechanism generates a pulse on the appropriate input (Q, D, or N). To signify the occurrence of the event, the mechanism also generates a pulse on the line *Coin*. The circuit is reset by using the *Resetn* signal (active low). When at least 30 cents has been deposited, the circuit activates its output, Z. No change is given if the amount exceeds 30 cents.

Design the required circuit by using the following components: a six-bit adder, a six-bit register, and any number of AND, OR, and NOT gates.

Solution: Figure 5.68 gives a possible circuit. The value of each coin is represented by a corresponding five-bit number. It is added to the current total, which is held in register *S*. The required output is

$$Z = s_5 + s_4 s_3 s_2 s_1$$

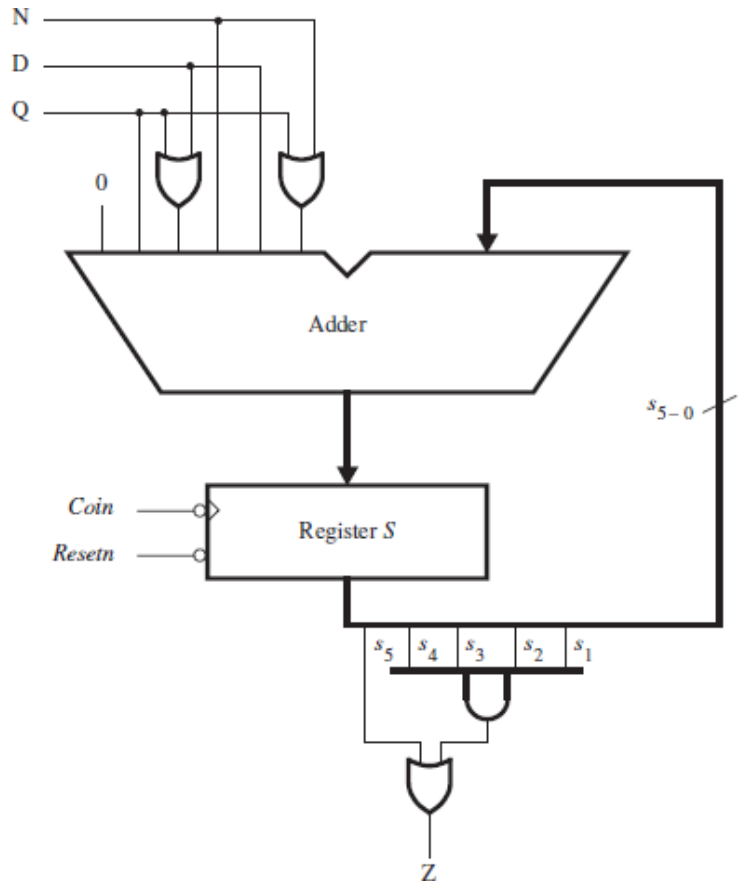


Figure 5.68 Circuit for Example 5.16.

The register is clocked by the negative edge of the *Coin* signal. This allows for a propagation delay through the adder, and ensures that a correct sum will be placed into the register.

In Chapter 9 we will show how this type of control circuit can be designed using a more structured approach.

Example 5.17

Problem: Write VHDL code to implement the circuit in Figure 5.68.

Solution: Figure 5.69 gives the desired code.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY vend IS
    PORT ( N, D, Q, Resetn, Coin  : IN   STD_LOGIC ;
           Z                      : OUT  STD_LOGIC ) ;
END vend ;

ARCHITECTURE Behavior OF vend IS
    SIGNAL X:  STD_LOGIC_VECTOR(4 DOWNTO 0) ;
    SIGNAL S:  STD_LOGIC_VECTOR(5 DOWNTO 0) ;
BEGIN
    X(0) <= N OR Q ;
    X(1) <= D ;
    X(2) <= N ;
    X(3) <= D OR Q ;
    X(4) <= Q ;
    PROCESS ( Resetn, Coin )
    BEGIN
        IF Resetn = '0' THEN
            S <= "000000" ;
        ELSIF Coin'EVENT AND Coin = '0' THEN
            S <= ('0' & X) + S ;
        END IF ;
    END PROCESS ;
    Z <= S(5) OR (S(4) AND S(3) AND S(2) AND S(1)) ;
END Behavior ;
```

Figure 5.69 Code for Example 5.17.

Example 5.18

Problem: In Section 5.15 we presented a timing analysis for the counter circuit in Figure 5.62. Redesign this circuit to reduce the logic delay between flip-flops, so that the circuit can operate at a higher maximum clock frequency.

Solution: As we showed in Section 5.15, the performance of the counter circuit is limited by the delay through its cascaded AND gates. To increase the circuit's performance we can refactor the AND gates as illustrated in Figure 5.70. We assume that the delay through a logic gate is 1 + 0.1*k*, where *k* is the number gate inputs. The longest delay path in this redesigned circuit, which starts at flip-flop Q_0 and ends at Q_3, provides the minimum clock period

$$T_{min} = t_{cQ} + t_{AND} + t_{XOR} + t_{su}$$
$$= 1.0 + 1.4 + 1.2 + 0.6 \text{ ns} = 4.2 \text{ ns}$$

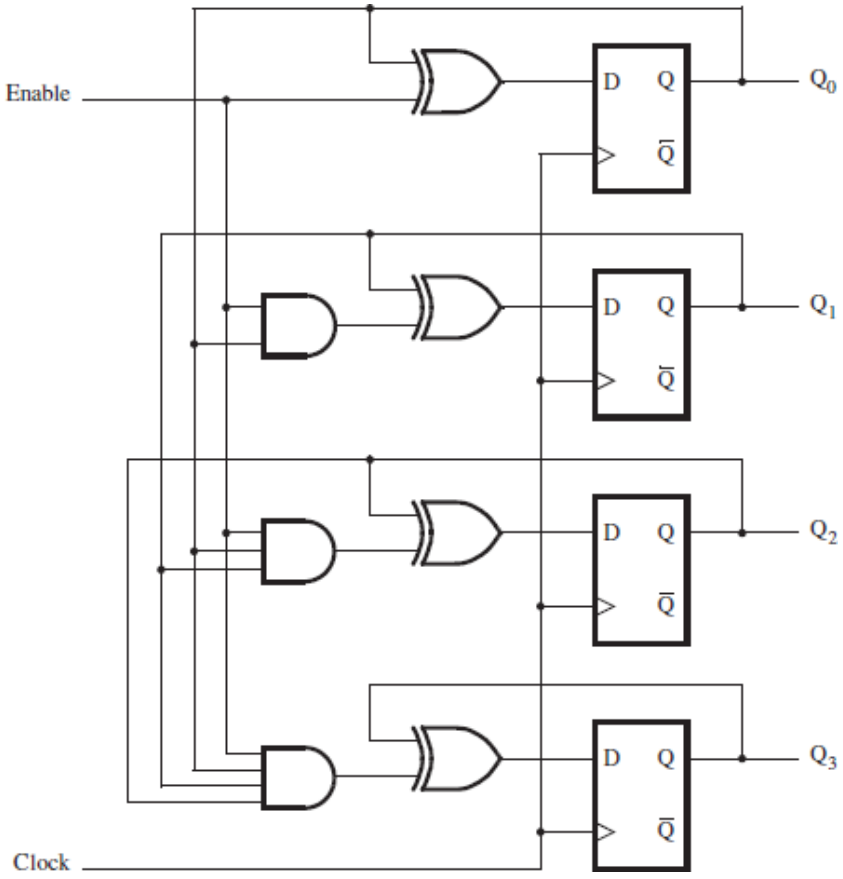


Figure 5.70 A faster 4-bit counter.

The redesigned counter has a maximum clock frequency of F_{max}=1/4.2 ns = 238.1 MHz, compared to the result for the original counter, which was 156.25 MHz.

Example 5.19

Problem: In Section 5.15 we showed how to perform timing analysis when there may be different delays associated with the clock signal at each of the flip-flops

in a circuit. The circuit in Figure 5.71 includes three flip-flops, Q_1, Q_2, and Q_3, with corresponding clock delays Δ_1, Δ_2, and Δ_3. The flip-flop timing parameters are $t_{su} = 0.6$ ns, $t_h = 0.4$ ns, and $0.8 \le t_{cQ} \le 1$ ns. Also, the delay through a k-input logic gate is $1 + 0.1k$.

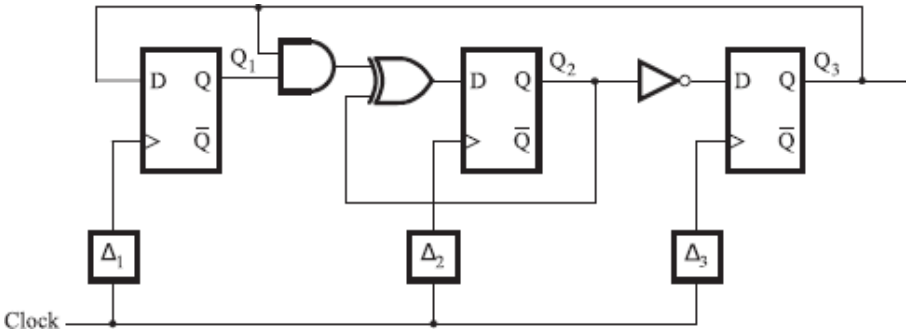


Figure 5.71 A circuit with clock skews.

Calculate the F_{max} for the circuit for the following sets of clock delays: $\Delta_1 = \Delta_2 = \Delta_3 = 0$ ns; $\Delta_1 = \Delta_3 = 0$ ns and $\Delta_2 = 0.7$ ns; $\Delta_1 = 1$ ns, $\Delta_2 = 0$, and $\Delta_3 = 0.5$ ns. Also, determine if there are any hold time violations in the circuit for the sets of clock delays $\Delta_1 = \Delta_2 = \Delta_3 = 0$ ns, and $\Delta_1 = 1$ ns, $\Delta_2 = 0$, $\Delta_3 = 0.5$ ns.

Solution: For the values $\Delta_1 = \Delta_2 = \Delta_3 = 0$, there is no clock skew. Let the path delay between two flip-flops, Q_i and Q_j, be $T_{Q_i \to Q_j}$. The longest path delays are calculated by including the maximum value of t_{cQ} at flip-flop Q_i, and the setup time at flip-flop Q_j, as follows

$$T_{Q_1 \to Q_2} = t_{cQ} + t_{XOR} + t_{AND} + t_{su} = 1 + 1.2 + 1.2 + 0.6 = 4 \text{ ns}$$
$$T_{Q_2 \to Q_2} = t_{cQ} + t_{AND} + t_{su} = 1 + 1.2 + 0.6 = 2.8 \text{ ns}$$
$$T_{Q_2 \to Q_3} = t_{cQ} + t_{NOT} + t_{su} = 1 + 1.1 + 0.6 = 2.7 \text{ ns}$$
$$T_{Q_3 \to Q_1} = t_{cQ} + t_{su} = 1 + 0.6 = 1.6 \text{ ns}$$
$$T_{Q_3 \to Q_2} = t_{cQ} + t_{XOR} + t_{AND} + t_{su} = 1 + 1.2 + 1.2 + 0.6 = 4 \text{ ns}$$

Since the critical path delay $T_{Q_1 \to Q_2} = T_{Q_3 \to Q_2} = 4$ ns, then $F_{max} = 1/(4 \times 10^{-9}) = 250$ MHz.

For the values $\Delta_1 = \Delta_3 = 0$, and $\Delta_2 = 0.7$ ns, there is clock skew for the paths between flip-flops Q_1 and Q_2, as well as Q_3 and Q_2. Adjusting the longest delays calculated above to account for clock skew, we have

$$T_{Q_1 \to Q_2} = 4 - t_{skew} = 4 - (\Delta_2 - \Delta_1) = 4 - 0.7 = 3.3 \text{ ns}$$
$$T_{Q_2 \to Q_3} = 2.7 - t_{skew} = 2.7 - (\Delta_3 - \Delta_2) = 2.7 - (0 - 0.7) = 3.4 \text{ ns}$$
$$T_{Q_3 \to Q_2} = 4 - t_{skew} = 4 - (\Delta_2 - \Delta_3) = 4 - 0.7 = 3.3 \text{ ns}$$

The critical path delay now starts at flip-flop Q_2 and ends at flip-flop Q_3. Since $T_{Q_2 \to Q_3} = 3.4$ ns, then $F_{max} = 1/(3.4 \times 10^{-9}) = 294$ MHz.

For the values $\Delta_1 = 1$, $\Delta_2 = 0$, and $\Delta_3 = 0.5$ ns, there is clock skew for the paths between all flip-flops. Adjusting the longest delays calculated for the case with no clock skew, we have

$$
\begin{aligned}
T_{Q_1 \to Q_2} &= 4 - t_{skew} = 4 - (\Delta_2 - \Delta_1) = 4 - (0 - 1) = 5 \text{ ns} \\
T_{Q_2 \to Q_3} &= 2.7 - t_{skew} = 2.7 - (\Delta_3 - \Delta_2) = 2.7 - (0.5) = 2.2 \text{ ns} \\
T_{Q_3 \to Q_1} &= 1.6 - t_{skew} = 1.6 - (\Delta_1 - \Delta_3) = 1.6 - (1 - 0.5) = 1.1 \text{ ns} \\
T_{Q_3 \to Q_2} &= 4 - t_{skew} = 4 - (\Delta_2 - \Delta_3) = 4 - (0 - 0.5) = 4.5 \text{ ns}
\end{aligned}
$$

The critical path delay is from flip-flop Q_1 to flip-flop Q_2. Since $T_{Q_1 \to Q_2} = 5$ ns, then $F_{max} = 1/(5 \times 10^{-9}) = 200$ MHz.

To determine whether any hold time violations exist, we need to calculate the shortest path delays, using the minimum value of t_{cQ}. For $\Delta_1 = \Delta_2 = \Delta_3 = 0$, we have

$$
\begin{aligned}
T_{Q_1 \to Q_2} &= t_{cQ} + t_{XOR} + t_{AND} = 0.8 + 1.2 + 1.2 = 3.2 \text{ ns} \\
T_{Q_2 \to Q_2} &= t_{cQ} + t_{AND} = 0.8 + 1.2 = 2 \text{ ns} \\
T_{Q_2 \to Q_3} &= t_{cQ} + t_{NOT} = 0.8 + 1.1 = 1.9 \text{ ns} \\
T_{Q_3 \to Q_1} &= t_{cQ} = 0.8 \text{ ns} \\
T_{Q_3 \to Q_2} &= t_{cQ} + t_{XOR} + t_{AND} = 0.8 + 1.2 + 1.2 = 3.2 \text{ ns}
\end{aligned}
$$

Since the shortest path delay $T_{Q_3 \to Q_1} = 0.8 \text{ ns} > t_h$, there is no hold time violation in the circuit.

We showed in Section 5.15 that if the shortest path delay through a circuit is called T_l, then a hold time violation occurs if $T_l - t_{skew} < t_h$. Adjusting the shortest path delays calculated above for the values $\Delta_1 = 1$, $\Delta_2 = 0$, and $\Delta_3 = 0.5$ gives

$$
\begin{aligned}
T_{Q_1 \to Q_2} &= 3.2 - t_{skew} = 3.2 - (\Delta_2 - \Delta_1) = 3.2 - (0 - 1) = 4.2 \text{ ns} \\
T_{Q_2 \to Q_3} &= 1.9 - t_{skew} = 1.9 - (\Delta_3 - \Delta_2) = 1.9 - 0.5 = 1.4 \text{ ns} \\
T_{Q_3 \to Q_1} &= 0.8 - t_{skew} = 0.8 - (\Delta_1 - \Delta_3) = 0.8 - (1 - 0.5) = 0.3 \text{ ns} \\
T_{Q_3 \to Q_2} &= 3.2 - t_{skew} = 3.2 - (\Delta_2 - \Delta_3) = 3.2 - (0 - 0.5) = 3.7 \text{ ns}
\end{aligned}
$$

The shortest path delay is $T_{Q_3 \to Q_1} = 0.3 \text{ ns} < t_h$, which represents a hold time violation. Hence, the circuit may not function reliably regardless of the frequency of the clock signal.

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

5.1 Consider the timing diagram in Figure P5.1. Assuming that the *D* and *Clock* inputs shown are applied to the circuit in Figure 5.10, draw waveforms for the Q_a, Q_b, and Q_c signals.

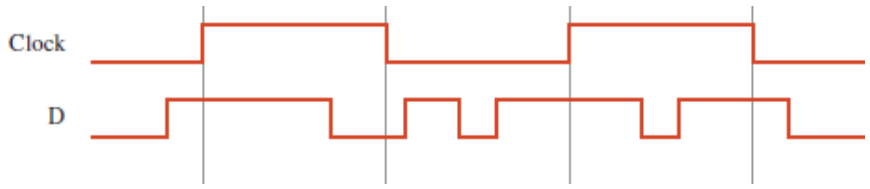


Figure P5.1 Timing diagram for Problem 5.1.

5.2 Figure 5.4 shows a latch built with NOR gates. Draw a similar latch using NAND gates. Derive its characteristic table and show its timing diagram.

***5.3** Show a circuit that implements the gated SR latch using NAND gates only.

5.4 Given a 100-MHz clock signal, derive a circuit using D flip-flops to generate 50-MHz and 25-MHz clock signals. Draw a timing diagram for all three clock signals, assuming reasonable delays.

***5.5** An SR flip-flop is a flip-flop that has set and reset inputs like a gated SR latch. Show how an SR flip-flop can be constructed using a D flip-flop and other logic gates.

5.6 The gated SR latch in Figure 5.5*a* has unpredictable behavior if the *S* and *R* inputs are both equal to 1 when the *Clk* changes to 0. One way to solve this problem is to create a *set-dominant* gated SR latch in which the condition $S = R = 1$ causes the latch to be set to 1. Design a set-dominant gated SR latch and show the circuit.

5.7 Show how a JK flip-flop can be constructed using a T flip-flop and other logic gates.

***5.8** Consider the circuit in Figure P5.2. Assume that the two NAND gates have much longer (about four times) propagation delay than the other gates in the circuit. How does this circuit compare with the circuits that we discussed in this chapter?

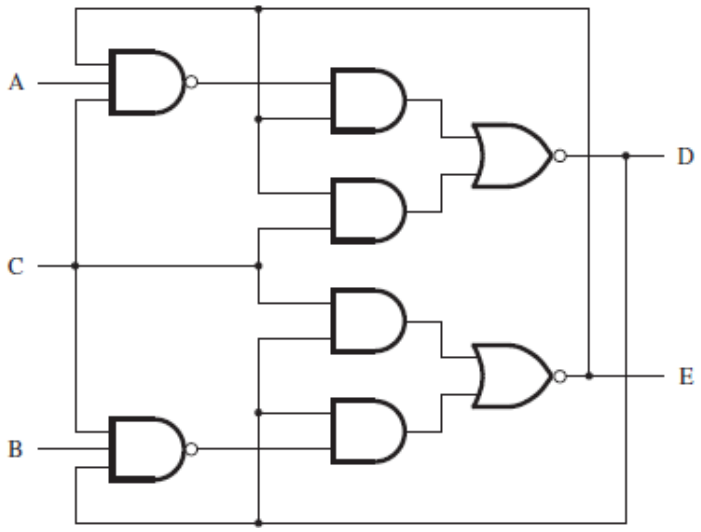


Figure P5.2 Circuit for Problem 5.8.

5.9 Write VHDL code that represents a T flip-flop with an asynchronous clear input. Use behavioral code, rather than structural code.

5.10 Write VHDL code that represents a JK flip-flop. Use behavioral code, rather than structural code.

5.11 Synthesize a circuit for the code written for Problem 5.10 by using your CAD tools. Simulate the circuit and show a timing diagram that verifies the desired functionality.

5.12 A universal shift register can shift in both the left-to-right and right-to-left directions, and it has parallel-load capability. Draw a circuit for such a shift register.

5.13 Write VHDL code for a universal shift register with *n* bits.

5.14 Design a four-bit synchronous counter with parallel load. Use T flip-flops, instead of the D flip-flops used in Section 5.9.3.

***5.15** Design a three-bit up/down counter using T flip-flops. It should include a control input called $\overline{\text{Up}}/\text{Down}$. If $\overline{\text{Up}}/\text{Down} = 0$, then the circuit should behave as an up-counter. If $\overline{\text{Up}}/\text{Down} = 1$, then the circuit should behave as a down-counter.

5.16 Repeat Problem 5.15 using D flip-flops.

***5.17** The circuit in Figure P5.3 looks like a counter. What is the counting sequence of this circuit?

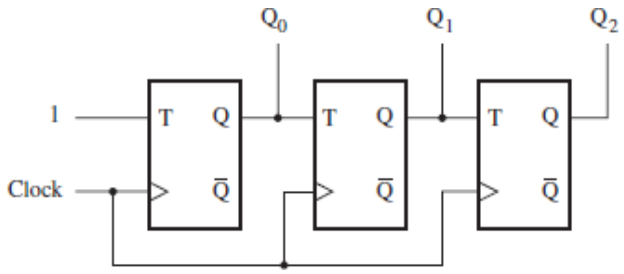


Figure P5.3 Circuit for Problem 5.17.

5.18 Consider the circuit in Figure P5.4. How does this circuit compare with the circuit in Figure 5.16? Can the circuits be used for the same purposes? If not, what is the key difference between them?

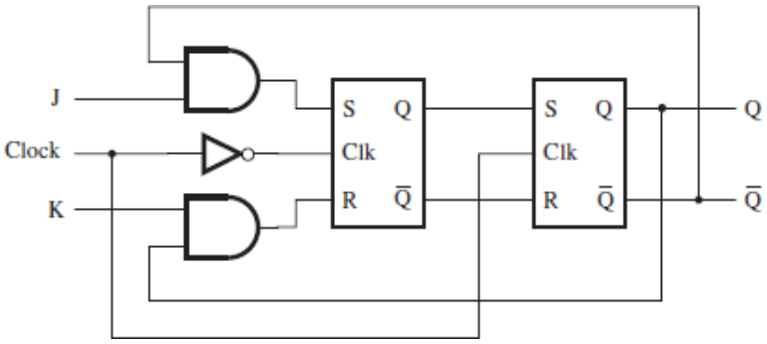


Figure P5.4 Circuit for Problem 5.18.

5.19 Construct a NOR-gate circuit, similar to the one in Figure 5.11*a*, which implements a negative-edge-triggered D flip-flop.

5.20 Write behavioral VHDL code that represents a 24-bit up/down-counter with parallel load and asynchronous reset.

5.21 Modify the VHDL code in Figure 5.48 by adding a parameter that sets the number of flip-flops in the counter.

5.22 Write behavioral VHDL code that represents a modulo-12 up-counter with synchronous reset.

***5.23** For the flip-flops in the counter in Figure 5.24, assume that t_{su} = 3 ns, t_h = 1 ns, and the propagation delay through a flip-flop is 1 ns. Assume that each AND gate, XOR gate, and 2-to-1 multiplexer has the propagation delay equal to 1 ns. What is the maximum clock frequency for which the circuit will operate correctly?

5.24 Write hierarchical code (structural) for the circuit in Figure 5.27. Use the counter in Figure 5.24 as a subcircuit.

5.25 Write VHDL code that represents an eight-bit Johnson counter. Use a simulation tool to verify the counting sequence.

5.26 Write behavioral VHDL code in the style shown in Figure 5.47 that represents a ring counter. Your code should have a parameter *N* that sets the number of flip-flops in the counter.

5.27 A ring oscillator is a circuit that has an odd number, *n*, of inverters connected in a ringlike structure, as shown in Figure P5.5. The output of each inverter is a periodic signal with a certain period.

(a) Assume that all the inverters are identical; hence they all have the same delay, called t_p. Let the output of one of the inverters be named *f*. Give an equation that expresses the period of the signal *f* in terms of *n* and t_p.

(b) For this part you are to design a circuit that can be used to experimentally measure the delay t_p through one of the inverters in the ring oscillator. Assume the existence of an input called *Reset* and another called *Interval*. The timing of these two signals is shown in Figure P5.6. The length of time for which *Interval* has the value 1 is known. Assume that this length of time is 100 ns. Design a circuit that uses the *Reset* and *Interval* signals and the signal *f* from part (*a*) to experimentally measure t_p. In your design you may use logic gates and subcircuits such as adders, flip-flops, counters, registers, and so on.

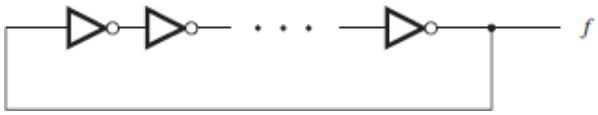


Figure P5.5 A ring oscillator.

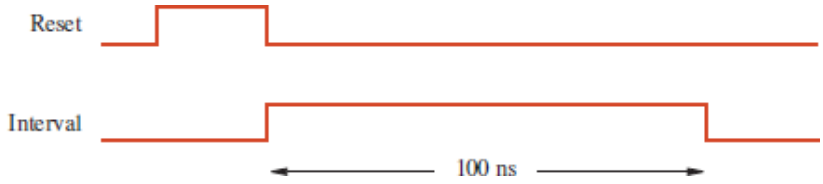


Figure P5.6 Timing signals for Problem 5.27.

5.28 A circuit for a gated D latch is shown in Figure P5.7. Assume that the propagation delay through either a NAND gate or an inverter is 1 ns. Complete the timing diagram given in the figure, which shows the signal values with 1 ns resolution.

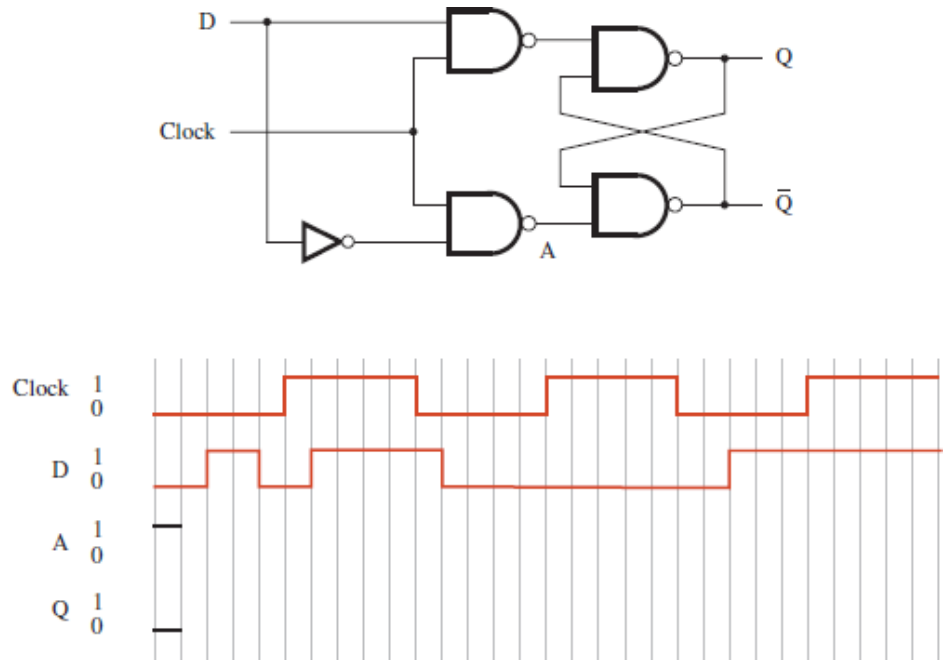


Figure P5.7 Circuit and timing diagram for Problem 5.28.

***5.29** A logic circuit has two inputs, *Clock* and *Start*, and two outputs, *f* and *g*. The behavior of the circuit is described by the timing diagram in Figure P5.8. When a pulse is received on the *Start* input, the circuit produces pulses on the *f* and *g* outputs as shown in the timing diagram. Design a suitable circuit using only the following components: a three-bit resettable positive-edge-triggered synchronous counter and basic logic gates. For your answer assume that the delays through all logic gates and the counter are negligible.

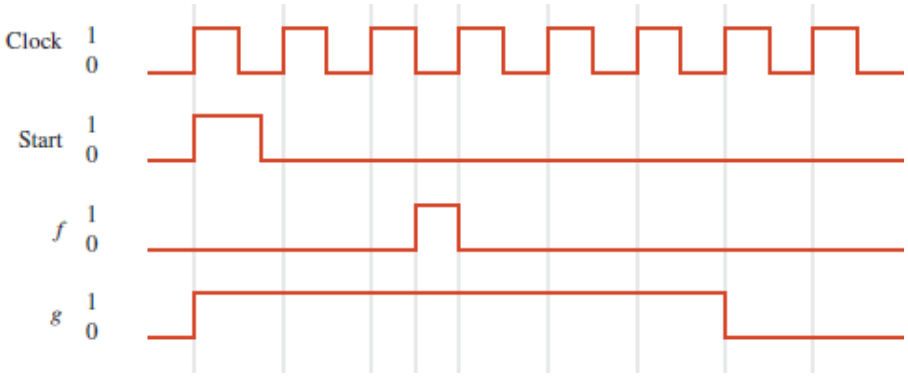


Figure P5.8 Timing diagram for Problem 5.29.

5.30 Write behavioral VHDL code for a four-digit BCD counter.

5.31 Determine the maximum clock frequency that can be used for the circuit in Figure 5.24. Use the timing parameters given in Section 5.15.

5.32 (a) Draw a circuit that could be synthesized from the VHDL code in Figure P5.9.

(b) How would you modify this code to specify a crossbar switch? In the crossbar switch each output y_j can be driven by either input x_i.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY problem IS
    PORT ( x1, x2, s: IN     STD_LOGIC ;
           y1, y2   : OUT  STD_LOGIC) ;
END problem ;

ARCHITECTURE Behavior OF problem IS
BEGIN
    PROCESS ( x1, x2, s )
    BEGIN
        IF s = '0' THEN
            y1 <= x1 ;
            y2 <= x2 ;
        ELSIF s = '1' THEN
            y1 <= x2 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure P5.9 Code for Problem 5.32.

5.33 A digital control circuit has three inputs: *Start*, *Stop*, and *Clock*, as well as an output signal *Run*. The *Start* and *Stop* signals are of indeterminate duration and may span many clock cycles. When the *Start* signal goes to 1, the circuit must generate $Run = 1$. The *Run* signal must remain high until the *Stop* signal goes to 1, at which time it has to return to 0. All changes in the *Run* signal must be synchronized with the *Clock* signal.

(a) Design the desired control circuit.

(b) Write VHDL code that specifies the desired circuit.

chapter

6

SYNCHRONOUS SEQUENTIAL CIRCUITS

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Design techniques for circuits that use flip-flops
- The concept of states and their implementation with flip-flops
- Synchronous control by using a clock signal
- Sequential behavior of digital circuits
- A complete procedure for designing synchronous sequential circuits
- VHDL specification of sequential circuits
- The concept of finite state machines

In preceding chapters we considered combinational logic circuits in which outputs are determined fully by the present values of inputs. We also discussed how simple storage elements can be implemented in the form of

flip-flops. The output of a flip-flop depends on the state of the flip-flop rather than the value of its inputs at any given time; the inputs cause changes in the state.

In this chapter we deal with a general class of circuits in which the outputs depend on the past behavior of the circuit, as well as on the present values of inputs. They are called *sequential circuits*. In most cases a clock signal is used to control the operation of a sequential circuit; such a circuit is called a *synchronous sequential circuit*. The alternative, in which no clock signal is used, is called an *asynchronous sequential circuit*. Synchronous circuits are easier to design and are used in a vast majority of practical applications; they are the topic of this chapter. Asynchronous circuits will be discussed in Chapter 9.

Synchronous sequential circuits are realized using combinational logic and one or more flip-flops. The general structure of such a circuit is shown in Figure 6.1. The circuit has a set of primary inputs, *W*, and produces a set of outputs, *Z*. The stored values in the flip-flops are referred to as the *state*, Q, of the circuit. Under control of the clock signal, the flip-flops change their state as determined by the combinational logic that feeds the inputs of these flip-flops. Only one transition from one state to another can take place during a single clock cycle because of the edge-triggered nature of the flip-flops. They can be triggered either by the positive (0 to 1 transition) or by the negative (1 to 0 transition) edge of the clock. We will use the term *active clock edge* to refer to the clock edge that causes the change in state.

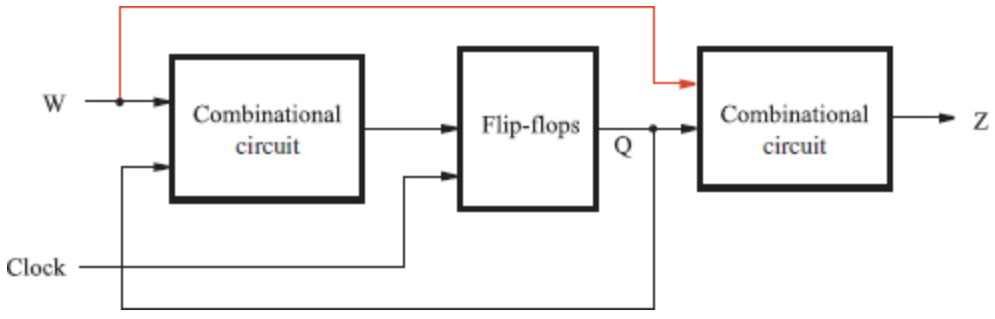


Figure 6.1 The general form of a sequential circuit.

The combinational logic that provides the input signals to the flip-flops has two sources: the primary inputs, *W*, and the present (current) state of the flip-flops, Q. Thus changes in state depend on both the present state and the values of the primary inputs.

Figure 6.1 indicates that the outputs of the sequential circuit are generated by another combinational circuit, such that the outputs are a function of the present state of the flip-flops and of the primary inputs. Although the outputs always depend on the present state, they do not necessarily have to depend directly on the primary inputs. Thus the connection shown in red in the figure may or may not exist. To distinguish between these two possibilities, it is customary to say that sequential circuits whose outputs depend only on the state of the circuit are of *Moore* type, while those whose outputs depend on both the state and the primary inputs are of *Mealy* type. These names are in honor of Edward Moore and George Mealy, who investigated the behavior of such circuits in the 1950s.

Sequential circuits are also called *finite state machines (FSMs)*, which is a more formal name that is often found in technical literature. The name derives from the fact that the functional behavior of these circuits can be represented using a finite number of states. In this chapter we will often use the term *finite state machine*, or simply *machine*, when referring to sequential circuits.

6.1 Basic Design Steps

Sequential circuits are often used to control the operation of physical systems. We will introduce the techniques for designing such circuits by means of a simple example.

Consider an application where the speed of an automatically controlled vehicle has to be regulated as follows. The vehicle is designed to run at some predetermined speed. However, due to some operational conditions the speed may exceed the desirable limit, in which case the vehicle has to be slowed down. To determine when such action is needed, the speed is measured at regular intervals. Let a binary signal w indicate whether the speed exceeds the required limit, such that $w = 0$ means that the speed is within an acceptable range and $w = 1$ indicates excessive speed. The desired control strategy is that if $w = 1$ during two or more consecutive measurements, a control signal z must be asserted to cause the vehicle to slow down. Thus, $z = 0$ allows the current speed to be maintained, while $z = 1$ reduces the speed. Let a signal *Clock* define the required timing intervals,

such that the speed is measured once during each clock cycle. Therefore, we wish to design a circuit that meets the following specification:

1. The circuit has one input, w, and one output, z.
2. All changes in the circuit occur on the positive edge of the clock signal.
3. The output z is equal to 1 if during two immediately preceding clock cycles the input w was equal to 1. Otherwise, the value of z is equal to 0.

From this specification it is apparent that the output z depends on both present and past values of w. Consider the sequence of values of the w and z signals for the clock cycles shown in Figure 6.2. The values of w are assumed to be generated by the vehicle being controlled; the values of z correspond to our specification. Since $w = 1$ in clock cycle t_3 and remains equal to 1 in cycle t_4, then z should be set to 1 in cycle t_5. Similarly, z should be set to 1 in cycle t_8 because $w = 1$ in cycles t_6 and t_7, and it should also be set to 1 in t_9 because $w = 1$ in t_7 and t_8. As seen in the figure, for a given value of w the output z may be either 0 or 1. For example, $w = 0$ during clock cycles t_2 and t_5, but $z = 0$ during t_2 and $z = 1$ during t_5. Similarly, $w = 1$ during t_1 and t_8, but $z = 0$ during t_1 and $z = 1$ during t_8. This means that z is not determined only by the present value of w, so there must exist different states in the circuit that determine the value of z.

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0

Figure 6.2 Sequences of input and output signals.

6.1.1 STATE DIAGRAM

The first step in designing a finite state machine is to determine how many states are needed and which transitions are possible from one state to another. There is no set procedure for this task. The designer must think carefully about what the machine has to accomplish. A good way to begin is to select one particular state as a *starting* state; this is the state that the circuit should enter when power is first turned on or when a *reset* signal is applied. For our example let us assume that the starting state is called state

A. As long as the input *w* is 0, the circuit need not do anything, and so each active clock edge should result in the circuit remaining in state *A*. When *w* becomes equal to 1, the machine should recognize this and move to a different state, which we will call state *B*. This transition takes place on the next active clock edge after *w* has become equal to 1. In state *B*, as in state *A*, the circuit should keep the value of output *z* at 0, because it has not yet seen $w = 1$ for two consecutive clock cycles. When in state *B*, if *w* is 0 at the next active clock edge, the circuit should move back to state *A*. However, if $w = 1$ when in state *B*, the circuit should change at the next active clock edge to a third state, called *C*, and it should then generate an output $z = 1$. The circuit should remain in state *C* as long as $w = 1$ and should continue to maintain $z = 1$. When *w* becomes 0, the machine should move back to state *A*. Since the preceding description handles all possible values of the input *w* that the machine can encounter in its various states, we can conclude that three states are needed to implement the desired machine.

Now that we have determined in an informal way the possible transitions between states, we will describe a more formal procedure that can be used to design the corresponding sequential circuit. Behavior of a sequential circuit can be described in several different ways. The conceptually simplest method is to use a pictorial representation in the form of a *state diagram*, which is a graph that depicts states of the circuit as nodes (circles) and transitions between states as directed arcs. The state diagram in Figure 6.3 defines the behavior that corresponds to our specification. States *A*, *B*, and *C* appear as nodes in the diagram. Node *A* represents the starting state, and it is also the state that the circuit will reach *after* an input $w = 0$ is applied. In this state the output *z* should be 0, which is indicated as $A/z = 0$ in the node. The circuit should remain in state *A* as long as $w = 0$, which is indicated by an arc with a label $w = 0$ that originates and terminates at this node. The first occurrence of $w = 1$ (following the condition $w = 0$) is recorded by moving from state *A* to state *B*. This transition is indicated on the graph by an arc originating at *A* and terminating at *B*. The label $w = 1$ on this arc denotes the input value that causes the transition. In state *B* the output remains at 0, which is indicated as $B/z = 0$ in the node.

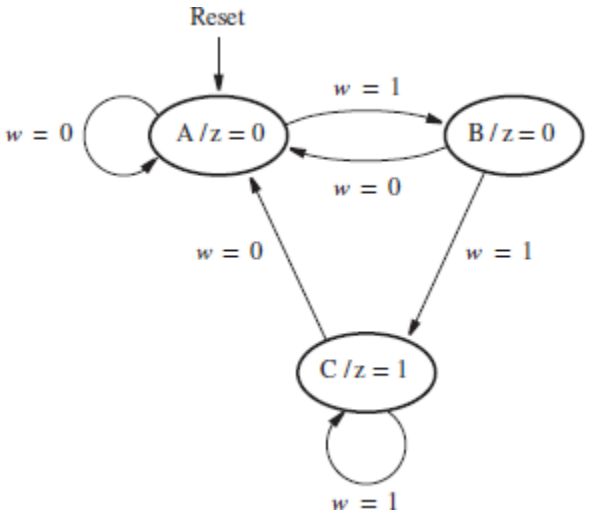


Figure 6.3 State diagram of a simple sequential circuit.

When the circuit is in state *B*, it will change to state *C* if *w* is still equal to 1 at the next active clock edge. In state *C* the output *z* becomes equal to 1. If *w* stays at 1 during subsequent clock cycles, the circuit will remain in state *C* maintaining $z = 1$. However, if *w* becomes 0 when the circuit is either in state *B* or in state *C*, the next active clock edge will cause a transition to state *A*.

In the diagram we indicated that the *Reset* input is used to force the circuit into state *A*. We could treat *Reset* as just another input to the circuit, and show a transition from each state to the starting state *A* under control of the input *Reset*. This would complicate the diagram unnecessarily. Instead, we use a single arrow with the *Reset* label, as shown in Figure 6.3, to indicate that regardless of what state the circuit happens to be in the *Reset* input causes a change to the starting state.

6.1.2 STATE TABLE

Although the state diagram provides a description of the behavior of a sequential circuit that is easy to understand, to proceed with the implementation of the circuit it is convenient to translate the information contained in the state diagram into a tabular form. Figure 6.4 shows the *state table* for our sequential circuit. The table indicates all transitions from each *present state* to the *next state* for different values of the input signal. Note that the output *z* is specified with respect to the present state, namely, the state that the circuit is in at present time. Note also that we did not

include the *Reset* input; instead, we made an implicit assumption that the first state in the table is the starting state.

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	A	B	0
B	A	C	0
C	A	C	1

Figure 6.4 State table corresponding to Figure 6.3.

We now show the design steps that will produce the final circuit. To explain the basic design concepts, we first go through a traditional process of manually performing each design step. This is followed by a discussion of automated design techniques that use modern computer aided design (CAD) tools.

6.1.3 STATE ASSIGNMENT

The state table in Figure 6.4 defines the three states in terms of letters *A*, *B*, and *C*. When implemented in a logic circuit, each state is represented by a particular valuation (combination of values) of *state variables*. Each state variable may be implemented using a flip-flop. Since three states have to be realized, it is sufficient to use two state variables. Let these variables be y_1 and y_2.

Now we can adapt the general block diagram in Figure 6.1 to our example as shown in Figure 6.5, to indicate the structure of the circuit that implements the required finite state machine. Two flip-flops represent the state variables. In the figure we have not specified the type of flip-flops to be used; this issue is addressed in the next subsection. From the specification in Figures 6.3 and 6.4, the output z is determined only by the present state of the circuit. Thus the block diagram in Figure 6.5 shows that z is a function of only y_1 and y_2; our circuit is of Moore type. We need to design a combinational circuit that uses y_1 and y_2 as input signals and generates a correct output signal z for all possible valuations of these inputs.

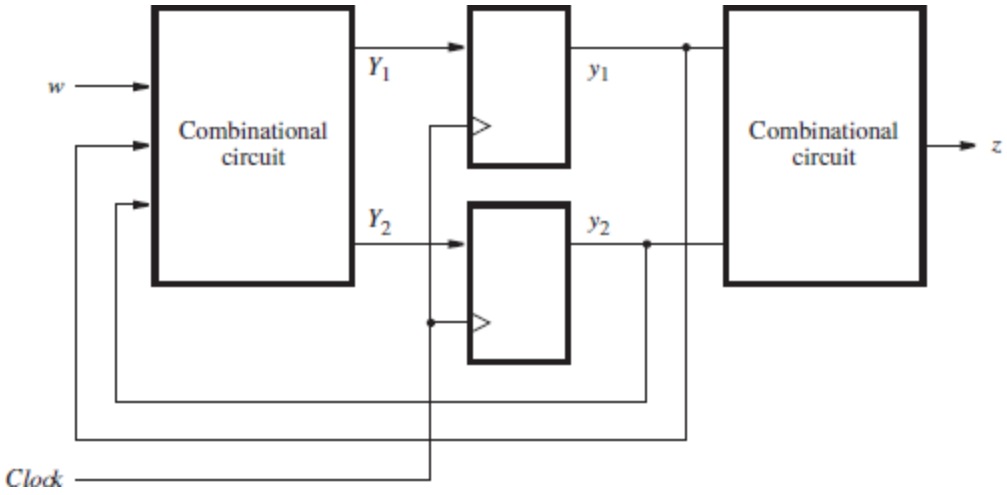


Figure 6.5 A sequential circuit with input *w*, output *z*, and two state flip-flops.

The signals y_1 and y_2 are also fed back to the combinational circuit that determines the next state of the FSM. This circuit also uses the primary input signal *w*. Its outputs are two signals, Y_1 and Y_2, which are used to set the state of the flip-flops. Each active edge of the clock will cause the flip-flops to change their state according to the values of Y_1 and Y_2 at that time. Therefore, Y_1 and Y_2 are called the *next-state variables*, and y_1 and y_2 are called the *present-state variables*. We need to design a combinational circuit with inputs *w*, y_1, and y_2, such that for all valuations of these inputs the outputs Y_1 and Y_2 will cause the machine to move to the next state that satisfies our specification. The next step in the design process is to create a truth table that defines this circuit, as well as the circuit that generates *z*.

To produce the desired truth table, we assign a specific valuation of variables y_1 and y_2 to each state. One possible assignment is given in Figure 6.6, where the states *A*, *B*, and *C* are represented by y_2y_1 = 00, 01, and 10, respectively. The fourth valuation, y_2y_1 = 11, is not needed in this case.

	Present state	Next state		Output
		$w = 0$	$w = 1$	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

Figure 6.6 State-assigned table corresponding to Figure 6.4.

The type of table given in Figure 6.6 is usually called a *state-assigned table*. This table can serve directly as a truth table for the output z with the inputs y_1 and y_2. Although for the next-state functions Y_1 and Y_2 the table does not have the appearance of a normal truth table, because there are two separate columns in the table for each value of w, it is obvious that the table includes all of the information that defines Y_1 and Y_2 in terms of valuations of inputs w, y_1, and y_2.

6.1.4 CHOICE OF FLIP-FLOPS AND DERIVATION OF NEXT-STATE AND OUTPUT EXPRESSIONS

From the state-assigned table in Figure 6.6, we can derive the logic expressions for the next-state and output functions. But first we have to decide on the type of flip-flops that will be used in the circuit. The most straightforward choice is to use D-type flip-flops, because in this case the values of Y_1 and Y_2 are simply clocked into the flip-flops to become the new values of y_1 and y_2. In other words, if the inputs to the flip-flops are called D_1 and D_2, then these signals are the same as Y_1 and Y_2. Note that the diagram in Figure 6.5 corresponds exactly to this use of D-type flip-flops. For other types of flip-flops, such as JK type, the relationship between the next-state variable and inputs to a flip-flop is not as straightforward; we will consider this situation in Section 6.7.

The required logic expressions can be derived as shown in Figure 6.7. We use Karnaugh maps to make it easy for the reader to verify the validity of the expressions. Recall that in Figure 6.6 we needed only three of the four possible binary valuations to represent the states. The fourth valuation, $y_2y_1 = 11$, will not occur in the circuit because the circuit is constrained to move only within states A, B, and C; therefore, we may choose to treat this valuation as a don't-care condition. The resulting don't-care squares in the Karnaugh maps are denoted by d's. Using the don't cares to simplify the expressions, we obtain

$$Y_1 = w\bar{y}_1\bar{y}_2$$
$$Y_2 = w(y_1 + y_2)$$
$$z = y_2$$

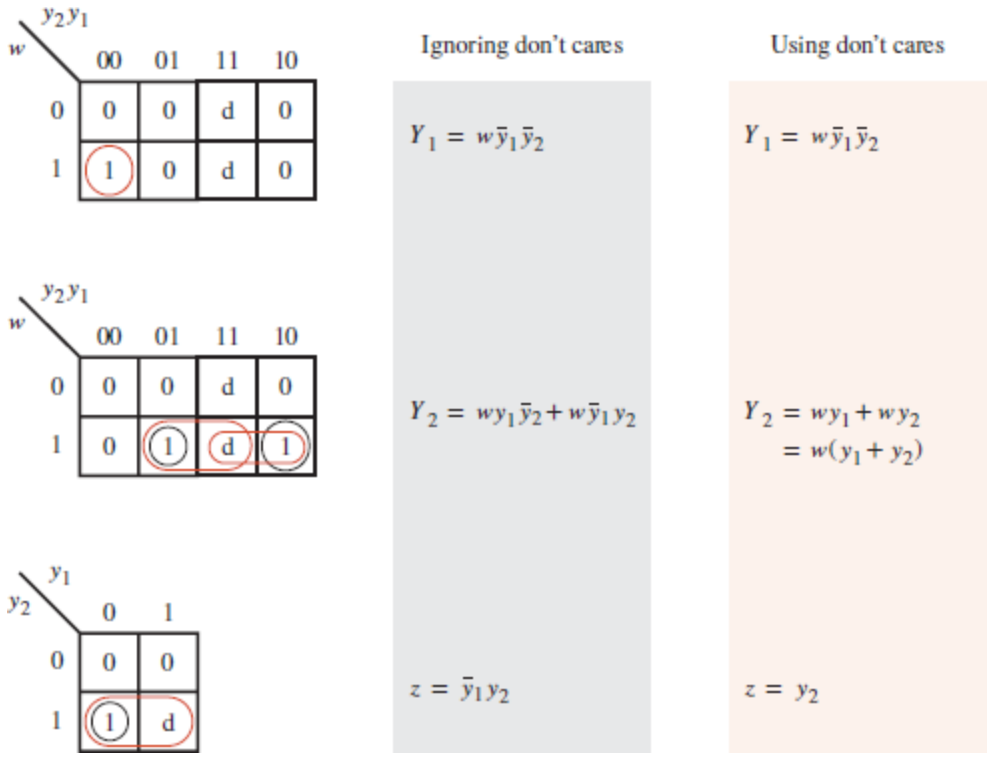


Figure 6.7 Derivation of logic expressions for the table in Figure 6.6.

If we do not use don't cares, then the resulting expressions are slightly more complex; they are shown in the gray-shaded area of Figure 6.7.

Since $D_1 = Y_1$ and $D_2 = Y_2$, the logic circuit that corresponds to the preceding expressions is implemented as shown in Figure 6.8. Observe that a clock signal is included, and the circuit is provided with an active-low reset capability. Connecting the clear input on the flip-flops to an external *Resetn* signal, as shown in the figure, provides a simple means for forcing the circuit into a known state. If we apply the signal *Resetn* = 0 to the circuit, then both flip-flops will be cleared to 0, placing the FSM into the state $y_2y_1 = 00$.

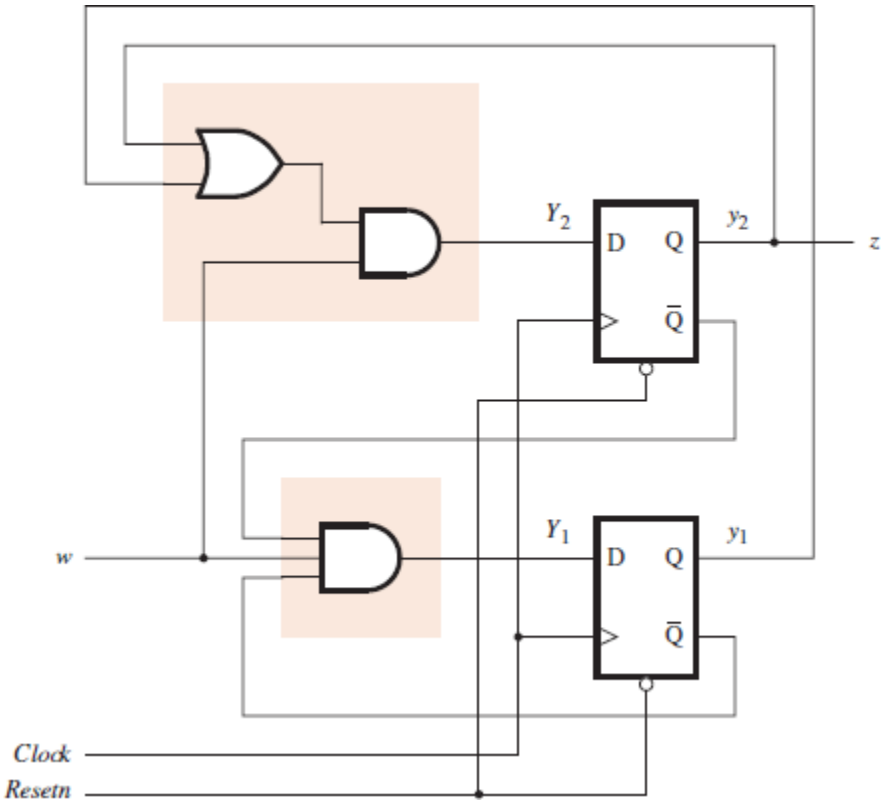


Figure 6.8 Final implementation of the sequential circuit.

6.1.5 TIMING DIAGRAM

To understand fully the operation of the circuit in Figure 6.8, let us consider its timing diagram presented in Figure 6.9. The diagram depicts the signal waveforms that correspond to the sequences of values in Figure 6.2.

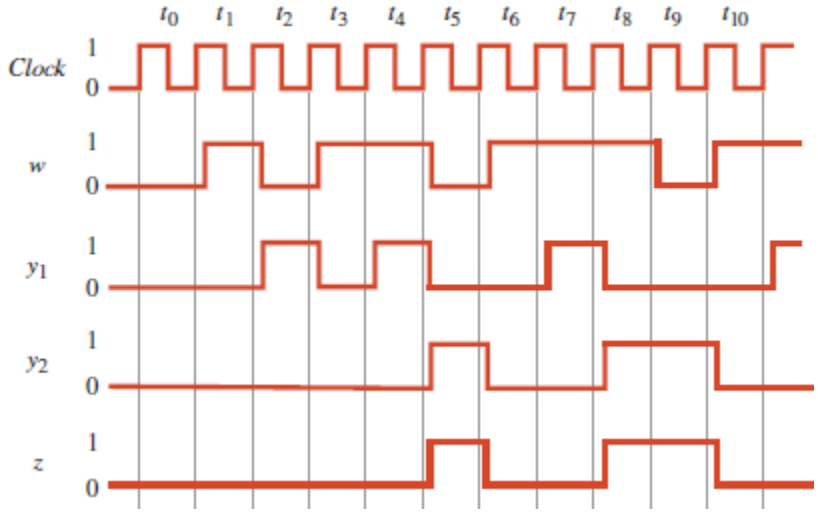


Figure 6.9 Timing diagram for the circuit in Figure 6.8.

Because we are using positive-edge-triggered flip-flops, all changes in the signals occur shortly after the positive edge of the clock. The amount of delay from the clock edge depends on the propagation delays through the flip-flops. Note that the input signal *w* is also shown to change slightly after the active edge of the clock. This is a good assumption because in a typical digital system an input such as *w* would be just an output of another circuit that is synchronized by the same clock. We discuss the synchronization of input signals with the clock signal in Chapter 7.

A key point to observe is that even though *w* changes slightly after the active clock edge, and thus the value of *w* is equal to 1 (or0) for almost the entire clock cycle, no change in the circuit will occur until the beginning of the next clock cycle when the positive edge causes the flip-flops to change their state. Thus the value of *w* must be equal to 1 for two clock cycles if the circuit is to reach state *C* and generate the output $z = 1$.

6.1.6 SUMMARY OF DESIGN STEPS

We can summarize the steps involved in designing a synchronous sequential circuit as follows:

1. Obtain the specification of the desired circuit.
2. Derive the states for the machine by first selecting a starting state. Then, given the specification of the circuit, consider all valuations of the inputs to the circuit and create new states as needed for the machine to respond to these inputs. To keep track of the states as they are visited, create a state diagram. When completed, the state diagram shows all states in the machine and gives the conditions under which the circuit moves from one state to another.
3. Create a state table from the state diagram. Alternatively, it may be convenient to directly create the state table in step 2, rather than first creating a state diagram.
4. In our sequential circuit example, there were only three states; hence it was a simple matter to create the state table that does not contain more states than necessary. However, in practice it is common to deal with circuits that have a large number of states. In such cases it is unlikely

that the first attempt at deriving a state table will produce optimal results, so that we may have more states than is really necessary. This can be corrected by a procedure that minimizes the number of states. We will discuss the process of state minimization in Section 6.6.

5. Decide on the number of state variables needed to represent all states and perform the state assignment. There are many different state assignments possible for a given sequential circuit. Some assignments may be better than others. In the preceding example we used what seemed to be a natural state assignment. We will return to this example in Section 6.2 and show that a different assignment may lead to a simpler circuit.
6. Choose the type of flip-flops to be used in the circuit. Derive the next-state logic expressions to control the inputs to all flip-flops and then derive logic expressions for the outputs of the circuit. So far we have used only D-type flip-flops. We will consider other types of flip-flops in Section 6.7.
7. Implement the circuit as indicated by the logic expressions.

We introduced the steps used in the design of sequential circuits by using an example of a simple controller for a vehicle. In this case the value of w reflected the speed of the vehicle. But, we could consider this example in a more general sense, in that the value of w in each clock cycle provides a *sequence* of symbols over time, where each symbol has the value 0 or 1. Our circuit can detect sequences of symbols that consist of consecutive 1s, and indicate this by generating $z = 1$ following each occurrence of two consecutive 1s. Since the circuit detects a specific sequence of symbols, we can say that it acts as a *sequence detector*. Similar circuits can be designed to detect a variety of different sequences.

Example 6.1

we have illustrated the design steps using a very simple sequential circuit. We will now consider a slightly more complex example that is closely tied to application in computer systems.

A computer system usually contains a number of registers that hold data during various operations. Sometimes it is necessary to swap the contents of two registers. Typically, this is done by using a temporary location, which is usually a third register. For example, suppose that we want to swap the contents of registers $R1$ and $R2$. We can achieve this by first transferring the contents of $R2$ into the third register, say $R3$. Next, we transfer the contents of $R1$ into $R2$. Finally, we transfer the contents of $R3$ into $R1$.

Registers in a computer system are connected via an interconnection network, as shown in Figure 6.10. In addition to the wires that connect to the network, each register has two control signals. The Rk_{out} signal causes the contents of register Rk to be placed into the interconnection network. The Rk_{in} signal causes the data from the network to be loaded into Rk. The Rk_{out} and Rk_{in} signals are generated by a control circuit, which is a finite state machine. For our example, we will design a control circuit that swaps the contents of $R1$ and $R2$, in response to an input $w = 1$. Therefore, the inputs to the control circuit will be w and *Clock*. The outputs will be $R1_{out}$, $R1_{in}$, $R2_{out}$, $R2_{in}$, $R3_{out}$, $R3_{in}$, and *Done* which indicates the completion of the required transfers.

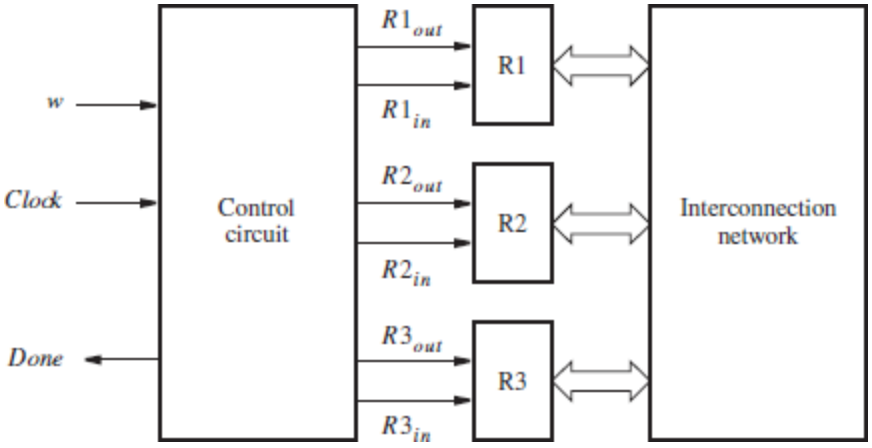


Figure 6.10 System for Example 6.1.

The desired swapping operation will be performed as follows. The contents of $R2$ are first loaded into $R3$, using the control signals $R2_{out} = 1$ and $R3_{in} = 1$. Then the contents of $R1$ are transferred into $R2$, using $R1_{out} = 1$ and $R2_{in} = 1$. Finally, the contents of $R3$ (which are the previous contents of $R2$) are transferred into $R1$, using $R3_{out} = 1$ and $R1_{in} = 1$. Since this step

completes the required swap, we will set the signal *Done* = 1. Assume that the swapping is performed in response to a pulse on the input *w*, which has a duration of one clock cycle. Figure 6.11 gives a state diagram for a sequential circuit that generates the output control signals in the required sequence. Note that to keep the diagram simple, we have indicated the output signals only when they are equal to 1. In all other cases the output signals are equal to 0.

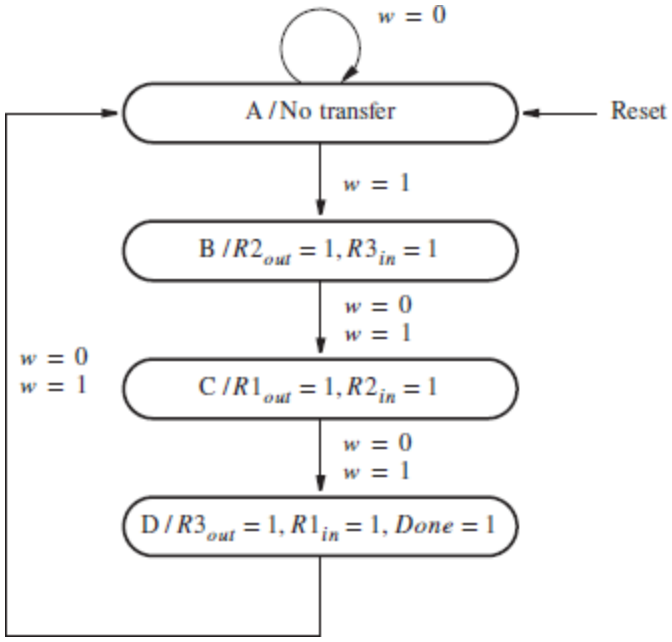


Figure 6.11 State diagram for Example 6.1.

In the starting state, *A*, no transfer is indicated, and all output signals are 0. The circuit remains in this state until a request to swap arrives in the form of *w* changing to 1. In state *B* the signals required to transfer the contents of *R*2 into *R*3 are asserted. The next active clock edge places these contents into *R*3. This clock edge also causes the circuit to change to state *C*, regardless of whether *w* is equal to 0 or 1. In this state the signals for transferring *R*1 into *R*2 are asserted. The transfer takes place at the next active clock edge, and the circuit changes to state *D* regardless of the value of *w*. The final transfer, from *R*3 to *R*1, is performed on the clock edge that leaves state *D*, which also causes the circuit to return to state *A*.

Figure 6.12 presents the same information in a state table. Since there are four states, we can use two state variables, y_2 and y_1. A straightforward state assignment where the states *A*, *B*, *C*, and *D* are assigned the valuations y_2y_1

= 00, 01, 10, and 11, respectively, leads to the state-assigned table in Figure 6.13. Using this assignment and D-type flip-flops, the next-state expressions can be derived as shown in Figure 6.14. They are

Present state	Next state		Outputs						
	$w = 0$	$w = 1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	A	B	0	0	0	0	0	0	0
B	C	C	0	0	1	0	0	1	0
C	D	D	1	0	0	1	0	0	0
D	A	A	0	1	0	0	1	0	1

Figure 6.12 State table for Example 6.1.

	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	0 0	0 0	0 1	0	0	0	0	0	0	0
B	0 1	1 0	1 0	0	0	1	0	0	1	0
C	1 0	1 1	1 1	1	0	0	1	0	0	0
D	1 1	0 0	0 0	0	1	0	0	1	0	1

Figure 6.13 State-assigned table corresponding to Figure 6.12.

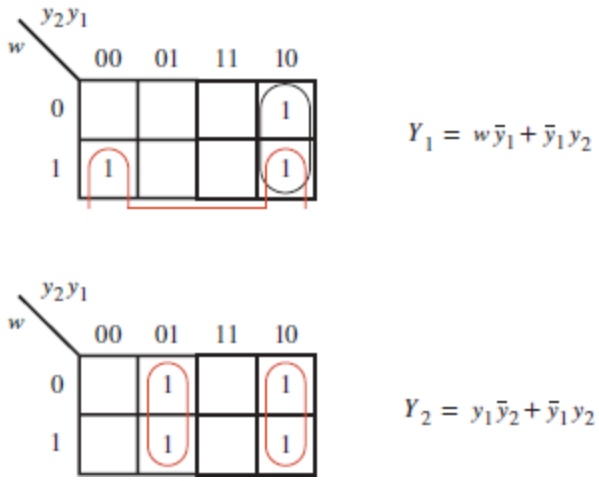


Figure 6.14 Derivation of next-state expressions for Figure 6.13.

$$Y_1 = w\bar{y}_1 + \bar{y}_1y_2$$
$$Y_2 = y_1\bar{y}_2 + \bar{y}_1y_2$$

The output control signals are derived as

$$R1_{out} = R2_{in} = \bar{y}_1 y_2$$
$$R1_{in} = R3_{out} = Done = y_1 y_2$$
$$R2_{out} = R3_{in} = y_1 \bar{y}_2$$

These expressions lead to the circuit in Figure 6.15.

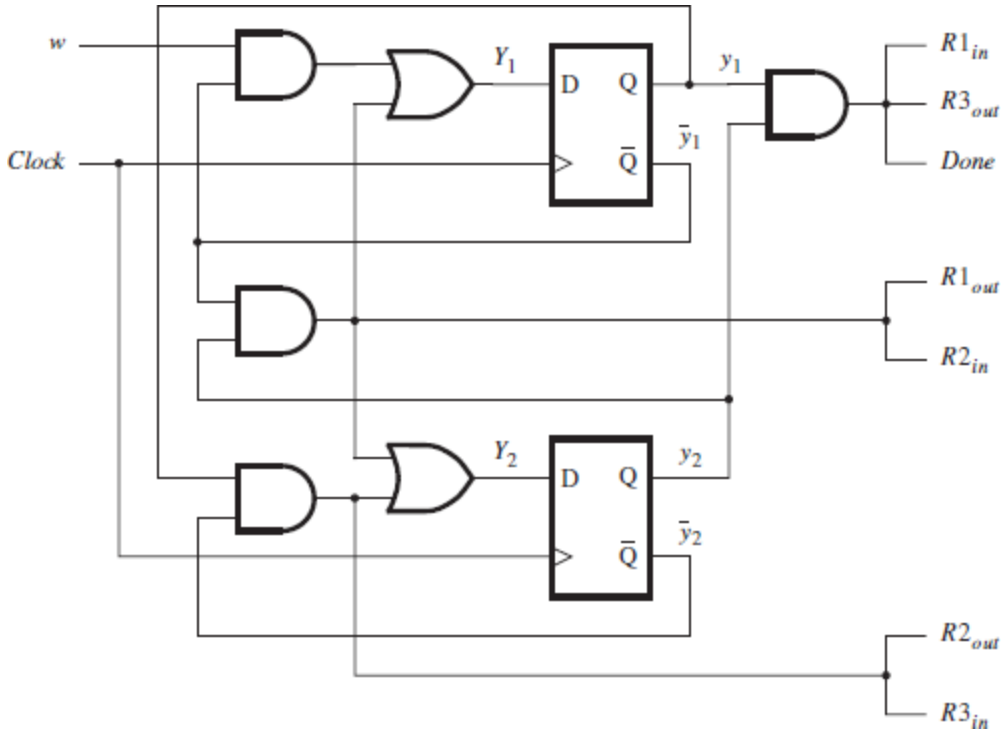


Figure 6.15 Final implementation of the sequential circuit for Example 6.1.

6.2 State-Assignment Problem

Having introduced the basic concepts involved in the design of sequential circuits, we should revisit some details where alternative choices are possible. In Section 6.1.6 we suggested that some state assignments may be better than others. To illustrate this we can reconsider the example in Figure 6.4. We already know that the state assignment in Figure 6.6 leads to a simple-looking circuit in Figure 6.8. But can the FSM of Figure 6.4 be

implemented with an even simpler circuit by using a different state assignment?

Figure 6.16 gives one possible alternative. In this case we represent the states *A*, *B*, and *C* with the valuations y_2y_1 = 00, 01, and 11, respectively. The remaining valuation, y_2y_1 = 10, is not needed, and we will treat it as a don't-care condition. The next-state and output expressions derived from the figure will be

	Present state	Next state		Output
		$w = 0$	$w = 1$	z
	y_2y_1	Y_2Y_1	Y_2Y_1	
A	00	00	01	0
B	01	00	11	0
C	11	00	11	1
	10	*dd*	*dd*	*d*

Figure 6.16 Improved state assignment for the state table in Figure 6.4.

$$Y_1 = D_1 = w$$
$$Y_2 = D_2 = wy_1$$
$$z = y_2$$

These expressions define the circuit shown in Figure 6.17. Comparing this circuit with the one in Figure 6.8, we see that the cost of the new circuit is lower because it requires fewer gates.

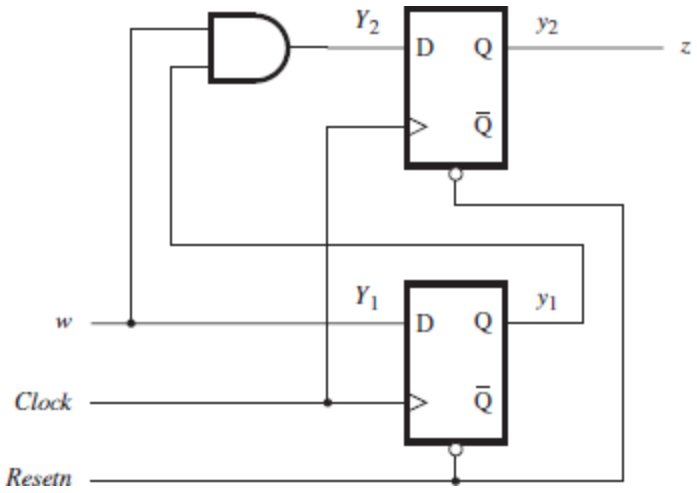


Figure 6.17 Final circuit for the improved state assignment in Figure 6.16.

In general, circuits are much larger than our example, and different state assignments can have a substantial effect on the cost of the final implementation. While highly desirable, it is often impossible to find the best state assignment for a large circuit. The exhaustive approach of trying all possible state assignments is not practical because the number of available state assignments is huge. CAD tools usually perform the state assignment using heuristic techniques. These techniques are usually proprietary, and their details are seldom published.

Example 6.2

In Figure 6.13 we used a straightforward state assignment for the sequential circuit in Figure 6.12. Consider now the effect of interchanging the valuations assigned to states *C* and *D*, as shown in Figure 6.18. Then the next-state expressions are

	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	*Done*
A	0 0	0 0	0 1	0	0	0	0	0	0	0
B	0 1	1 1	1 1	0	0	1	0	0	1	0
C	1 1	1 0	1 0	1	0	0	1	0	0	0
D	1 0	0 0	0 0	0	1	0	0	1	0	1

Figure 6.18 Improved state assignment for the state table in Figure 6.12.

$$Y_1 = w\bar{y}_2 + y_1\bar{y}_2$$
$$Y_2 = y_1$$

as derived in Figure 6.19. The output expressions are

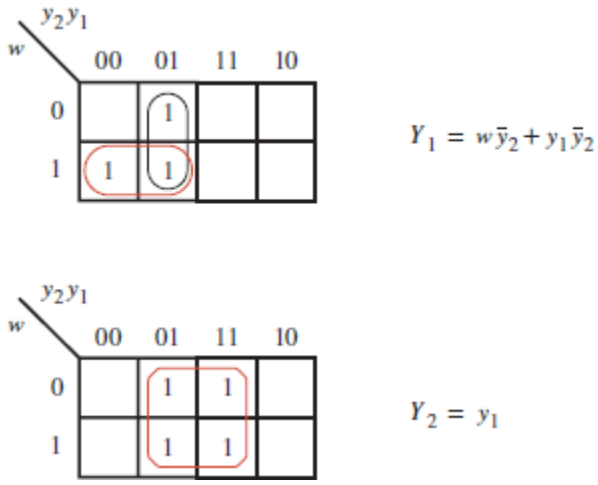


Figure 6.19 Derivation of next-state expressions for Figure 6.18.

$$R1_{out} = R2_{in} = y_1y_2$$
$$R1_{in} = R3_{out} = Done = \bar{y}_1y_2$$
$$R2_{out} = R3_{in} = y_1\bar{y}_2$$

These expressions lead to a slightly simpler circuit than the one given in Figure 6.15.

6.2.1 ONE-HOT ENCODING

In previous examples we used the minimum number of flip-flops to represent the states of the FSM. Another interesting possibility is to use as many state variables as there are states. In this method, for each state all but one of the state variables are equal to 0. The variable whose value is 1 is deemed to be "hot." The approach is known as the *one-hot encoding* method.

Figure 6.20 shows how one-hot state assignment can be applied to the sequential circuit of Figure 6.4. Because there are three states, it is necessary to use three state variables. The chosen assignment is to represent the states *A*, *B*, and *C* using the valuations $y_3y_2y_1$ = 001, 010, and 100, respectively. The remaining five valuations of the state variables are not used. They can be treated as don't cares in the derivation of the next-state and output expressions. Using this assignment, the resulting expressions are

$$Y_1 = \overline{w}$$
$$Y_2 = wy_1$$
$$Y_3 = w\overline{y}_1$$
$$z = y_3$$

	Present state	Next state		Output
		$w = 0$	$w = 1$	
	$y_3y_2y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$	z
A	001	001	010	0
B	010	001	100	0
C	100	001	100	1

Figure 6.20 One-hot state assignment for the state table in Figure 6.4.

Note that none of the next-state variables depends on the present-state variable y_2. This suggests that the second flip-flop and the expression $Y_2 = wy_1$ are not needed. (CAD tools detect and eliminate such redundancies!) But even then, the derived expressions are not simpler than those obtained using the state assignment in Figure 6.16. Although in this case the one-hot assignment is not advantageous, there are many cases where this approach is attractive.

Example 6.3

The one-hot state assignment can be applied to the sequential circuit of Figure 6.12 as indicated in Figure 6.21. Four state variables are needed, and the states *A*, *B*, *C*, and *D* are encoded as $y_4y_3y_2y_1$ = 0001, 0010, 0100, and 1000, respectively. Treating the remaining 12 valuations of the state variables as don't cares, the next-state expressions are

$$Y_1 = \overline{w}y_1 + y_4$$
$$Y_2 = wy_1$$
$$Y_3 = y_2$$
$$Y_4 = y_3$$

	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	$y_4y_3y_2y_1$	$Y_4Y_3Y_2Y_1$	$Y_4Y_3Y_2Y_1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	0001	0001	0010	0	0	0	0	0	0	0
B	0010	0100	0100	0	0	1	0	0	1	0
C	0100	1000	1000	1	0	0	1	0	0	0
D	1000	0001	0001	0	1	0	0	1	0	1

Figure 6.21 One-hot state assignment for the state table in Figure 6.12.

It is instructive to note that we can derive these expressions simply by inspecting the state diagram in Figure 6.11. There are two arcs leading to state *A* (not including the arc for the *Reset* signal). These arcs mean that flip-flop y_1 should be set to 1 if the FSM is already in state *A* and $w = 0$, or if the FSM is in state *D*; hence $Y_1 = \overline{w}y_1 + y_4$. Similarly, flip-flop y_2 should be set to 1 if the present state is *A* and $w = 1$; hence $Y_2 = wy_1$. Flip-flops y_3 and y_4 should be set to 1 if the FSM is presently in state *B* or *C*, respectively; hence $Y_3 = y_2$ and $Y_4 = y_3$.

The output expressions are just the outputs of the flip-flops, such that

$$R1_{out} = R2_{in} = y_3$$
$$R1_{in} = R3_{out} = Done = y_4$$
$$R2_{out} = R3_{in} = y_2$$

These expressions are simpler than those derived in Example 6.2, but four flip-flops are needed, rather than two.

An important feature of the one-hot state assignment is that it often leads to simpler output expressions than do assignments with the minimal number of state variables. Simpler output expressions may lead to a faster circuit. For instance, if the outputs of the sequential circuit are just the outputs of the flip-flops, as is the case in our example, then these output signals are valid as soon as the flip-flops change their states. If more complex output expressions are involved, then the propagation delay through the gates that implement these expressions must be taken into account. We will consider this issue in Section 6.8.2.

The examples considered to this point show that there are many ways to implement a given finite state machine as a sequential circuit. Each implementation is likely to have a different cost and different timing characteristics. In the next section we introduce another way of modeling FSMs that leads to even more possibilities.

6.3 Mealy State Model

Our introductory examples were sequential circuits in which each state had specific values of the output signals associated with it. As we explained at the beginning of the chapter, such finite state machines are said to be of Moore type. We will now explore the concept of Mealy-type machines in which the output values are generated based on both the state of the circuit and the present values of its inputs. This provides additional flexibility in the design of sequential circuits. We will introduce the Mealy-type machines, using a slightly altered version of a previous example.

The essence of the first sequential circuit in Section 6.1 is to generate an output $z = 1$ whenever a second occurrence of the input $w = 1$ is detected in consecutive clock cycles. The specification requires that the output z be equal to 1 in the clock cycle that follows the detection of the second occurrence of $w = 1$. Suppose now that we eliminate this latter requirement and specify instead that the output z should be equal to 1 in the same clock cycle when the second occurrence of $w = 1$ is detected. Then a suitable input-output sequence may be as shown in Figure 6.22. To see how we can realize the behavior given in this table, we begin by selecting a starting state, A. As long as $w = 0$, the machine should remain in state A, producing an output $z = 0$. When $w = 1$, the machine has to move to a new state, B, to record the fact that an input of 1 has occurred. If w remains equal to 1 when the machine is in state B, which happens if $w = 1$ for at least two consecutive clock cycles, the machine should remain in state B and produce an output $z = 1$. As soon as w becomes 0, z should immediately become 0 and the machine should move back to state A at the next active edge of the clock. Thus the behavior specified in Figure 6.22 can be achieved with a two-state machine, which has a state diagram shown in Figure 6.23. Only two states are needed because we have allowed the output value to depend

on the present value of the input as well as the present state of the machine. The diagram indicates that if the machine is in state *A*, it will remain in state *A* if $w = 0$ and the output will be 0. This is indicated by an arc with the label $w = 0/z = 0$. When w becomes 1, the output stays at 0 until the machine moves to state *B* at the next active clock edge. This is denoted by the arc from *A* to *B* with the label $w = 1/z = 0$. In state *B* the output will be 1 if $w = 1$, and the machine will remain in state *B*, as indicated by the label $w = 1/z = 1$ on the corresponding arc. However, if $w = 0$ in state *B*, then the output will be 0 and a transition to state *A* will take place at the next active clock edge. A key point to understand is that during the present clock cycle the output value corresponds to the label on the arc emanating from the present-state node.

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	1	0	0	1	1	0	0

Figure 6.22 Sequences of input and output signals.

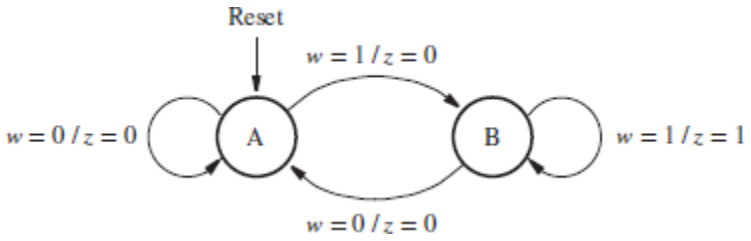


Figure 6.23 State diagram of an FSM that realizes the task in Figure 6.22.

We can implement the FSM in Figure 6.23, using the same design steps as in Section 6.1. The state table is shown in Figure 6.24. The table shows that the output z depends on the present value of input w and not just on the present state. Figure 6.25 gives the state-assigned table. Because there are only two states, it is sufficient to use a single state variable, y. Assuming that y is realized as a D-type flip-flop, the required next-state and output expressions are

$$Y = D = w$$
$$z = wy$$

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	A	B	0	1

Figure 6.24 State table for the FSM in Figure 6.23.

	Present state	Next state		Output	
		$w = 0$	$w = 1$	$w = 0$	$w = 1$
	y	Y	Y	z	z
A	0	0	1	0	0
B	1	0	1	0	1

Figure 6.25 State-assigned table for the FSM in Figure 6.24.

The resulting circuit is presented in Figure 6.26 along with a timing diagram. The timing diagram corresponds to the input-output sequences in Figure 6.22.

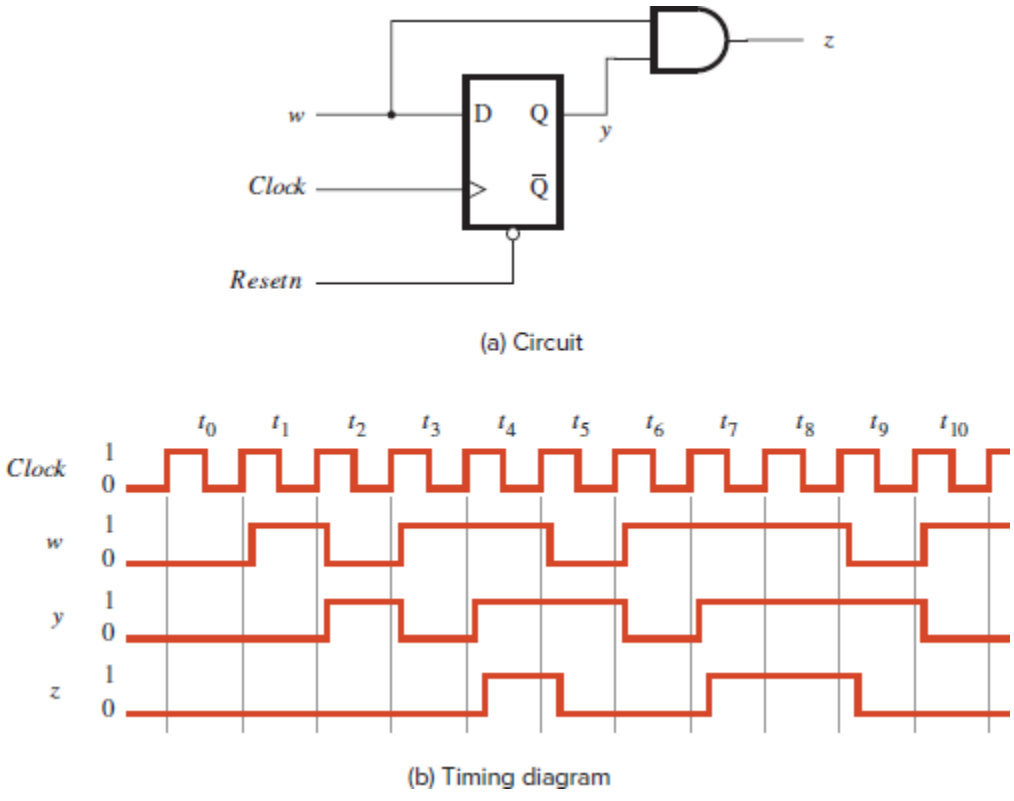
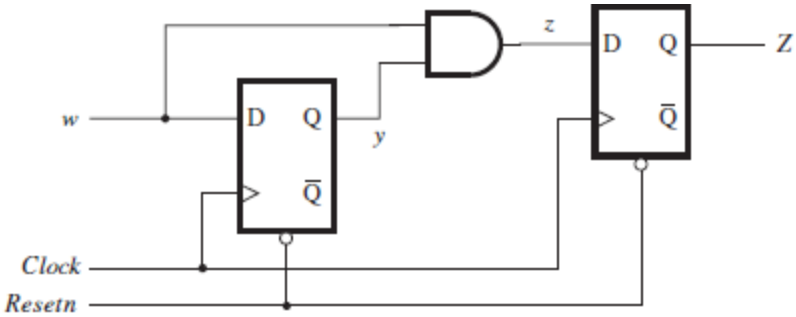
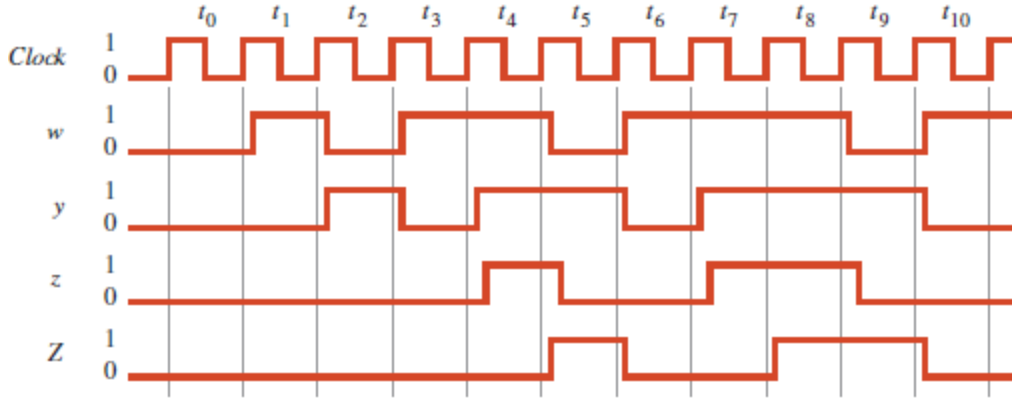


Figure 6.26 Implementation of the FSM in Figure 6.25.

The greater flexibility of Mealy-type FSMs often leads to simpler circuit realizations. This certainly seems to be the case in our examples that produced the circuits in Figures 6.8, 6.17, and 6.26, assuming that the design requirement is only to detect two consecutive occurrences of input *w* being equal to 1. We should note, however, that the circuit in Figure 6.26 is not the same in terms of output behavior as the circuits in Figures 6.8 and 6.17. The difference is a shift of one clock cycle in the output signal in Figure 6.26*b*. If we wanted to produce exactly the same output behavior using the Mealy approach, we could modify the circuit in Figure 6.26*a* by adding another flip-flop as shown in Figure 6.27. This flip-flop merely delays the output signal, *Z*, by one clock cycle with respect to *z*, as indicated in the timing diagram. By making this change, we effectively turn the Mealy-type circuit into a Moore-type circuit with output *Z*. Note that the circuit in Figure 6.27 is essentially the same as the circuit in Figure 6.17.



(a) Circuit



(b) Timing diagram

Figure 6.27 Circuit that implements the specification in Figure 6.2.

Example 6.4

In Example 6.1 we considered the control circuit needed to swap the contents of two registers, implemented as a Moore-type finite state machine. The same task can be achieved using a Mealy-type FSM, as indicated in Figure 6.28. State *A* still serves as the reset state. But as soon as *w* changes from 0 to 1, the output control signals $R2_{out}$ and $R3_{in}$ are asserted. They remain asserted until the beginning of the next clock cycle, when the circuit will leave state *A* and change to *B*. In state *B* the outputs $R1_{out}$ and $R2_{in}$ are asserted for both $w = 0$ and $w = 1$. Finally, in state *C* the swap is completed by asserting $R3_{out}$ and $R1_{in}$.

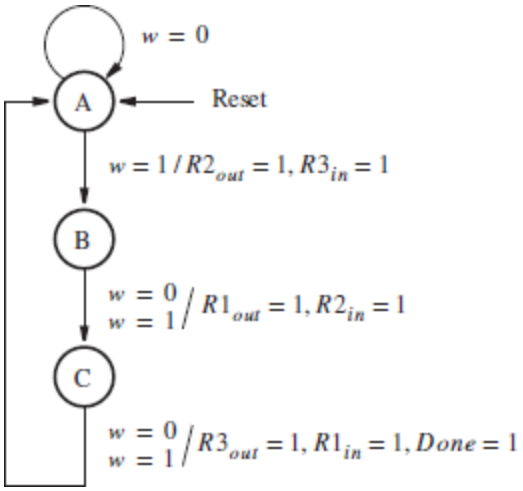


Figure 6.28 State diagram for Example 6.4.

The Mealy-type realization of the control circuit requires three states. This does not necessarily imply a simpler circuit because two flip-flops are still needed to implement the state variables. The most important difference in comparison with the Moore-type realization is the timing of output signals. A circuit that implements the FSM in Figure 6.28 generates the output control signals one clock cycle sooner than the circuits derived in Examples 6.1 and 6.2.

Note also that using the FSM in Figure 6.28, the entire process of swapping the contents of *R*1 and *R*2 takes three clock cycles, starting and finishing in state *A*. Using the Moore-type FSM in Example 6.1, the swapping process involves four clock cycles before the circuit returns to state *A*.

Suppose that we wish to implement this FSM using one-hot encoding. Then three flip-flops are needed, and the states A, B, and C may be assigned the valuations $y_3y_2y_1$ = 001, 010, and 100, respectively. Examining the state diagram in Figure 6.28, we can derive the next-state equations by inspection. The input to flip-flop y_1 should have the value 1 if the FSM is in state A and w = 0 or if the FSM is in state C; hence $Y_1 = \overline{w}y_1 + y_3$. Flip-flop y_2 should be set to 1 if the FSM is in state A and w = 1; hence $Y_2 = wy_1$. Flip-flop y_3 should be set to 1 if the present state is B; hence $Y_3 = y_2$. The derivation of the output expressions, which we leave as an exercise for the reader, can also be done by inspection.

The preceding discussion deals with the basic principles involved in the design of sequential circuits. Although it is essential to understand these principles, the manual approach used in the examples is difficult and tedious when large circuits are involved. We will now show how CAD tools are used to greatly simplify the design task.

6.4 Design of Finite State Machines Using CAD Tools

Sophisticated CAD tools are available for finite state machine design, and we introduce them in this section. A rudimentary way of using CAD tools for FSM design could be as follows: The designer employs the manual techniques described previously to derive a circuit that contains flip-flops and logic gates from a state diagram. This circuit is entered into the CAD system by drawing a schematic diagram or by writing structural hardware description language (HDL) code. The designer then uses the CAD system to simulate the behavior of the circuit and uses the CAD tools to automatically implement the circuit in a chip.

It is tedious to manually synthesize a circuit from a state diagram. Since CAD tools are meant to obviate this type of task, more attractive ways of utilizing CAD tools for FSM design have been developed. A better approach is to directly enter the state diagram into the CAD system and perform the entire synthesis process automatically. CAD tools support this

approach in two main ways. One method is to allow the designer to draw the state diagram using a graphical tool similar to the schematic capture tool. The designer draws circles to represent states and arcs to represent state transitions and indicates the outputs that the machine should generate. Another and more popular approach is to write HDL code that represents the state diagram, as described below.

Many HDLs provide constructs that allow the designer to represent a state diagram. To show how this is done, we will provide VHDL code that represents the simple machine designed manually as the first example in Section 6.1. Then we will use the CAD tools to synthesize a circuit that implements the machine in a chip.

6.4.1 VHDL CODE FOR MOORE-TYPE FSMS

VHDL does not define a standard way of describing a finite state machine. Hence while adhering to the required VHDL syntax, there is more than one way to describe a given FSM. An example of VHDL code for the FSM of Figure 6.3 is given in Figure 6.29. For the convenience of discussion, the lines of code are numbered on the left side. Lines 3 to 6 declare an entity named *simple*, which has input ports *Clock*, *Resetn*, and *w*, and output port *z*. In line 7 we have used the name *Behavior* for the architecture body, but of course, any valid VHDL name could be used instead.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY simple IS
4       PORT ( Clock, Resetn, w     : IN     STD_LOGIC ;
5              z                    : OUT  STD_LOGIC ) ;
6   END simple ;

7   ARCHITECTURE Behavior OF simple IS
8       TYPE State_type IS (A, B, C) ;
9       SIGNAL y : State_type ;
10  BEGIN
11      PROCESS ( Resetn, Clock )
12      BEGIN
13          IF Resetn = '0' THEN
14              y <= A ;
15          ELSIF (Clock'EVENT AND Clock = '1') THEN
16              CASE y IS
17                  WHEN A =>
18                      IF w = '0' THEN
19                          y <= A ;
20                      ELSE
21                          y <= B ;
22                      END IF ;
23                  WHEN B =>
24                      IF w = '0' THEN
25                          y <= A ;
26                      ELSE
27                          y <= C ;
28                      END IF ;
29                  WHEN C =>
30                      IF w = '0' THEN
31                          y <= A ;
32                      ELSE
33                          y <= C ;
34                      END IF ;
35              END CASE ;
36          END IF ;
37      END PROCESS ;
38      z <= '1' WHEN y = C ELSE '0' ;
39  END Behavior ;
```

Figure 6.29 VHDL code for the FSM in Figure 6.3.

Line 8 introduces the TYPE keyword, which is a feature of VHDL that we have not used previously. The TYPE keyword allows us to create a user-defined signal type. The new signal type is named State_type, and the code specifies that a signal of this type can have three possible values: *A*, *B*, or *C*. Line 9 defines a signal named *y* that is of the State_type type. The *y* signal

is used in the architecture body to represent the flip-flops that implement the states in the FSM. The code does not specify the number of bits represented by *y*. Instead, it specifies that *y* can have the three symbolic values *A*, *B*, and *C*. This means that we have not specified the number of state flip-flops that should be used for the FSM. As we will see below, the VHDL compiler automatically chooses an appropriate number of state flip-flops when synthesizing a circuit to implement the machine. It also chooses the state assignment for states *A*, *B*, and *C*. Some CAD systems assume that the first state listed in the TYPE statement (line 8) is the reset state for the machine. The state assignment that has all flip-flop outputs equal to 0 is used for this state. Later in this section, we will show how it is possible to manually specify the state encoding in the VHDL code if so desired.

Having defined a signal to represent the state flip-flops, the next step is to specify the transitions between states. Figure 6.29 gives one way to describe the state diagram. It is represented by the process in lines 11 to 37. The PROCESS statement describes the finite state machine as a sequential circuit. It is based on the approach we used to describe an edge-triggered D flip-flop in Section 5.12.2. The signals used by the process are *Resetn*, *Clock*, *w*, and *y*, and the only signal modified by the process is *y*. The input signals that can cause the process to change *y* are *Resetn* and *Clock*; hence these signals appear in the sensitivity list. The input *w* is not included in the sensitivity list because a change in the value of *w* cannot affect *y* until a change occurs in the *Clock* signal.

Lines 13 and 14 specify that the machine should enter state *A*, the reset state, if *Resetn* =0. Since the condition for the IF statement does not depend on the clock signal, the reset is asynchronous, which is why *Resetn* is included in the sensitivity list in line 11.

When the reset signal is not asserted, the ELSIF statement in line 15 specifies that the circuit depends on the positive edge of the clock signal. Observe that the ELSIF condition is the same as the condition we used to describe a positive-edge-triggered D flip-flop in Figure 5.38. The behavior of *y* is defined by the CASE statement in lines 16 to 35. It corresponds to the state diagram in Figure 6.3. Since the CASE statement is inside the ELSIF condition, any change in *y* can take place only as a result of a positive clock edge. In other words, the ELSIF condition implies that *y* must be implemented as the output of one or more flip-flops. Each WHEN clause in the CASE statement represents one state of the machine. For

example, the WHEN clause in lines 17 to 22 describes the machine's behavior when it is in state *A*. According to the IF statement beginning in line 18, when the FSM is in state *A*, if $w = 0$, the machine should remain in state *A*; but if $w = 1$, the machine should change to state *B*. The WHEN clauses in the CASE statement correspond exactly to the state diagram in Figure 6.3.

The final part of the state machine description appears in line 38. It specifies that if the machine is in state *C*, then the output *z* should be 1; otherwise, *z* should be 0.

6.4.2 SYNTHESIS OF VHDL CODE

To give an example of the circuit produced by a synthesis tool, we synthesized the code in Figure 6.29 for implementation in a chip. The synthesis resulted in two flip-flops, with inputs Y_1 and Y_2, and outputs y_1 and y_2. The next-state expressions generated by the synthesis tool are

$$Y_1 = w\bar{y}_1\bar{y}_2$$
$$Y_2 = wy_1 + wy_2$$

The output expression is

$$z = y_2$$

These expressions correspond to the case in Figure 6.7 when the unused state pattern $y_2\, y_1 = 11$ is treated as don't-cares in the Karnaugh maps for Y_1, Y_2, and *z*.

Figure 6.30 depicts a part of the FSM circuit implemented in a CPLD device. This type of device is described in Appendix B; it is a programmable chip that contains flip-flops and logic gates that can be interconnected in various ways by using programmable switches. Each of these switches is indicated in the figure by an ×. To keep the figure simple, only the logic resources in the chip used to implement y_1, y_2, and *z* are shown. The parts of the chip used for the circuit are highlighted in red.

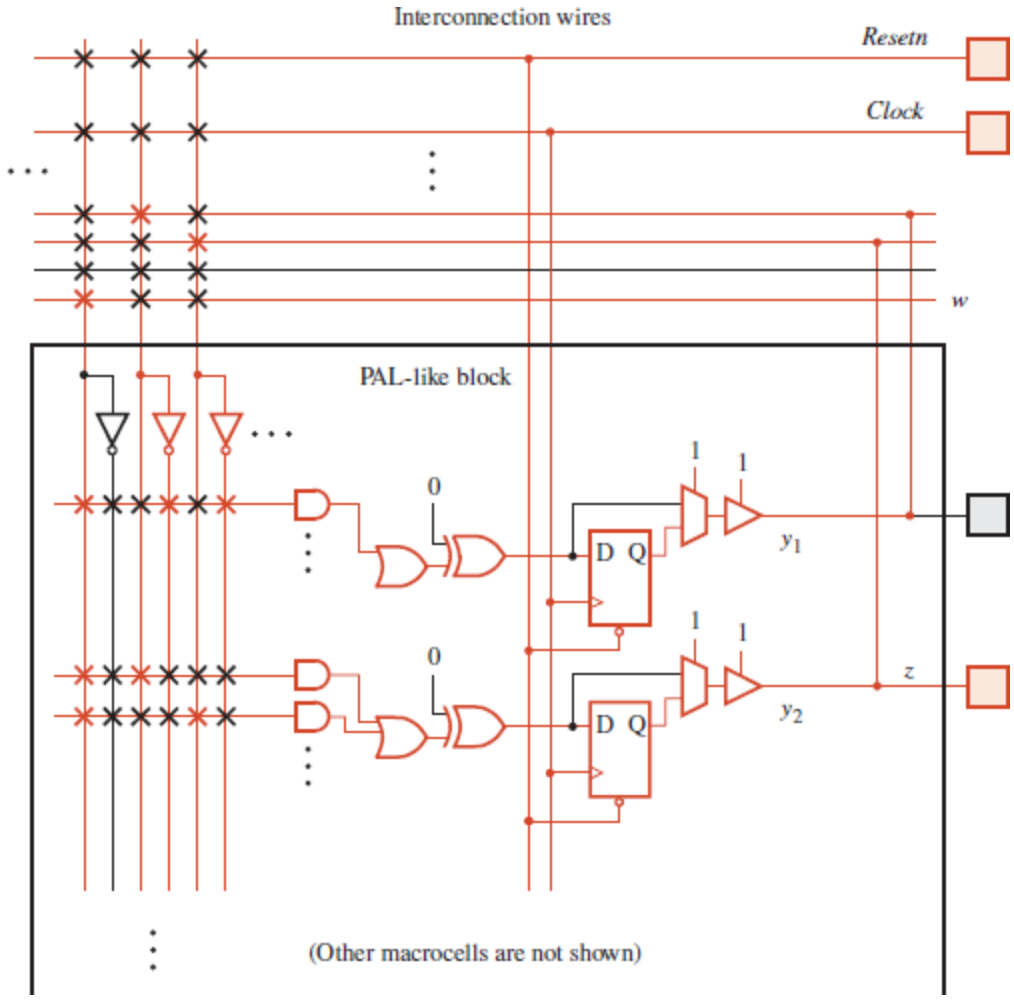


Figure 6.30 Implementation of the FSM of Figure 6.3 in a CPLD.

The *w* input to the circuit is shown connected to one of the interconnection wires in the CPLD. The source node in the chip that generates *w* is not shown. It could be either an input pin, or else *w* might be the output of another macrocell, assuming that the CPLD may contain other circuitry that is connected to our FSM. The *Clock* signal is assigned to a pin on the chip that is dedicated for use by clock signals. From this dedicated pin a *global wire* distributes the clock signal to all of the flip-flops in the chip. The global wire distributes the clock signal to the flip-flops such that the difference in the arrival time, or *clock skew*, of the clock signal at each flip-flop is minimized. The concept of clock skew is discussed in Chapter 5. A global wire is also used for the reset signal.

Figure 6.31 illustrates how the circuit might be assigned to the pins on a small CPLD. The figure is drawn with a part of the top of the chip package

cut away, revealing a conceptual view of the two macrocells from Figure 6.30, which are indicated in red. Our simple circuit uses only a small portion of the device.

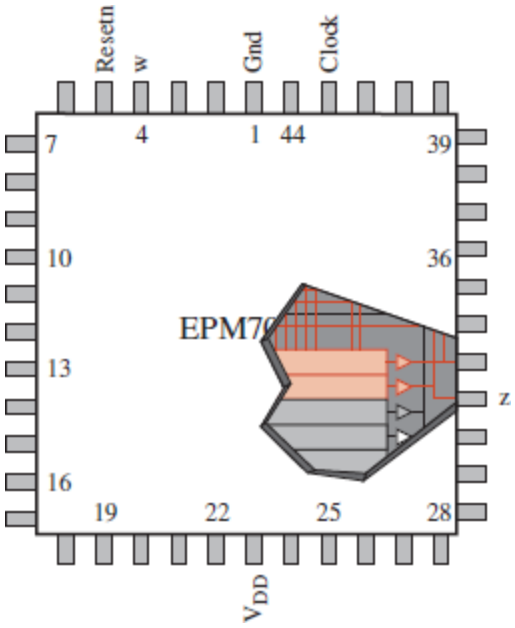


Figure 6.31 The circuit from Figure 6.30 in a small CPLD.

6.4.3 SIMULATING AND TESTING THE CIRCUIT

The behavior of the circuit implemented in the CPLD chip can be tested using simulation, as depicted in Figure 6.32. The figure gives the waveforms that correspond to the timing diagram in Figure 6.9, assuming that a 100 ns clock period is used. The *Resetn* signal is set to 0 at the beginning of the simulation and then set to 1. The circuit produces the output $z = 1$ for one clock cycle after w has been equal to 1 for two successive clock cycles. When w is 1 for three clock cycles, z becomes 1 for two clock cycles, as it should be. We show the changes in state by using the letters *A*, *B*, and *C* for readability purposes.

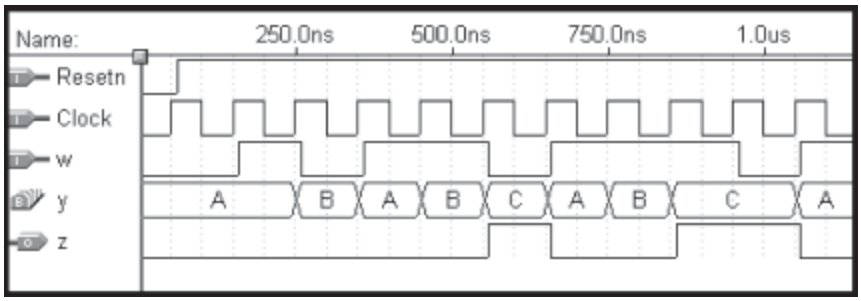


Figure 6.32 Simulation results for the circuit in Figure 6.30.

Having examined the simulation output, we should consider the question of whether we can conclude that the circuit functions correctly and satisfies all requirements. For our simple example it is not difficult to answer this question because the circuit has only one input and its behavior is straightforward. It is easy to see that the circuit works properly. However, in general it is difficult to ascertain with a high degree of confidence whether a sequential circuit will work properly for all possible input sequences, because a very large number of input patterns may be possible. For large finite state machines, the designer must think carefully about patterns of inputs that may be used in simulation for testing purposes.

6.4.4 An Alternative Style of VHDL Code

We mentioned earlier in this section that VHDL does not specify a standard way for writing code that represents a finite state machine. The code given in Figure 6.29 is only one possibility. A second example of code for our simple machine is given in Figure 6.33. Only the architecture body is shown because the entity declaration is the same as in Figure 6.29. Two signals are used to represent the state of the machine. The signal named *y_present* corresponds to the present state, and *y_next* corresponds to the next state. In terms of the notation used in Section 6.1.3, *y_present* is the same as *y*, and *y_next* is *Y*. We cannot use *y* to denote the present state and *Y* for the next state in the code, because VHDL does not distinguish between lowercase and uppercase letters. Both the *y_present* and *y_next* signals are of the State_type type.

```
ARCHITECTURE Behavior OF simple IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y_present, y_next : State_type ;
BEGIN
    PROCESS ( w, y_present )
    BEGIN
        CASE y_present IS
            WHEN A =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= B ;
                END IF ;
            WHEN B =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
            WHEN C =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
        END CASE ;
    END PROCESS ;

    PROCESS (Clock, Resetn)
    BEGIN
        IF Resetn = '0' THEN
            y_present <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            y_present <= y_next ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

Figure 6.33 Alternative style of VHDL code for the FSM in Figure 6.3.

The machine is specified by two separate processes. The first process describes the state table as a combinational circuit. It uses a CASE statement to give the value of *y_next* for each value of *y_present* and *w*. The code can be related to the general form of FSMs in Figure 6.5. The process corresponds to the combinational circuit on the left side of the figure.

The second process introduces flip-flops into the circuit. It stipulates that after each positive clock edge the *y_present* signal should take the value of

the *y_next* signal. The process also specifies that *y_present* should take the value *A* when *Resetn* =0, which provides the asynchronous reset.

We have shown two styles of VHDL code for our FSM example. The circuit produced by the VHDL compiler for each version of the code is likely to be somewhat different because, as the reader is well aware by this point, there are many ways to implement a given logic function. However, the circuits produced from the two versions of the code provide identical functionality.

6.4.5 Summary of Design Steps When Using CAD Tools

In Section 6.1.6 we summarized the design steps needed to derive sequential circuits manually. We have now seen that CAD tools can automatically perform much of the work. However, it is important to realize that the CAD tools have not replaced *all* manual steps. With reference to the list given in Section 6.1.6, the first two steps, in which the machine specification is obtained and a state diagram is derived, still have to be done manually. Given the state diagram information as input, the CAD tools then automatically perform the tasks needed to generate a circuit with logic gates and flip-flops. In addition to the design steps given in Section 6.1.6, we should add the testing and simulation stage. We will defer detailed discussion of this issue until Chapter 11.

6.4.6 Specifying the State Assignment in VHDL Code

In Section 6.2 we saw that the state assignment may have an impact on the complexity of the designed circuit. An obvious objective of the state-assignment process is to minimize the cost of implementation. The cost function that should be optimized may be simply the number of gates and flip-flops. But it could also be based on other considerations that may be representative of the type of chip used to implement the design.

In VHDL code it is possible to specify the state assignment that should be used, but there is no standardized way of doing so. Hence while adhering to VHDL syntax, each CAD system permits a slightly different method of specifying the state assignment. One way of performing state assignment is to use the *attribute* feature of VHDL. An attribute refers to some type of

information about an object in VHDL code. All signals automatically have a number of associated *predefined* attributes. An example is the EVENT attribute that we use to specify a clock edge, as in Clock'EVENT.

In addition to the predefined attributes, it is possible to create a user-defined attribute. The *user-defined* attribute can be used to associate some desired type of information with an object in VHDL code. Using this method, manual state assignment can be done by creating a user-defined attribute associated with the State_type type. This is illustrated in Figure 6.34, which shows the first few lines of the architecture from Figure 6.33 with the addition of a user-defined attribute. We first define the new attribute called ENUM_ENCODING, which has the type STRING. The next line associates ENUM_ENCODING with the State_type type and specifies that the attribute has the value "00 01 11". When translating the VHDL code, a CAD tool could use the value of ENUM_ENCODING to make the state assignment $A = 00$, $B = 01$, and $C = 11$.

```
ARCHITECTURE Behavior OF simple IS
    TYPE State_TYPE IS (A, B, C) ;
    ATTRIBUTE ENUM_ENCODING                 : STRING ;
    ATTRIBUTE ENUM_ENCODING OF State_type : TYPE IS "00 01 11" ;
    SIGNAL y_present, y_next                : State_type ;
BEGIN
    .
    .
    .
```

Figure 6.34 A user-defined attribute for manual state assignment.

The ENUM_ENCODING attribute is one method of specifying state codes, but this method may not be supported in some CAD systems. A different way of giving the state assignment, which will work with any CAD system, is shown in Figure 6.35.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY simple IS
    PORT ( Clock, Resetn, w  : IN   STD_LOGIC ;
           z                 : OUT  STD_LOGIC ) ;
END simple ;

ARCHITECTURE Behavior OF simple IS
    SIGNAL y_present, y_next : STD_LOGIC_VECTOR(1 DOWNTO 0);
    CONSTANT A : STD_LOGIC_VECTOR(1 DOWNTO 0) := "00" ;
    CONSTANT B : STD_LOGIC_VECTOR(1 DOWNTO 0) := "01" ;
    CONSTANT C : STD_LOGIC_VECTOR(1 DOWNTO 0) := "11" ;
BEGIN
    PROCESS ( w, y_present )
    BEGIN
        CASE y_present IS
            WHEN A =>
                IF w = '0' THEN y_next <= A ;
                ELSE y_next <= B ;
                END IF ;
            WHEN B =>
                IF w = '0' THEN y_next <= A ;
                ELSE y_next <= C ;
                END IF ;
            WHEN C =>
                IF w = '0' THEN y_next <= A ;
                ELSE y_next <= C ;
                END IF ;
            WHEN OTHERS =>
                y_next <= A ;
        END CASE ;
    END PROCESS ;
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            y_present <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            y_present <= y_next ;
        END IF ;
    END PROCESS ;
    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

Figure 6.35 Using constants for manual state assignment.

Instead of using the State_type type as in previous examples, *y_present* and *y_next* are defined as two-bit STD_LOGIC_VECTOR signals. Each of the symbolic names for the three states, *A*, *B*, and *C*, are defined as constants, with the value of each constant corresponding to the desired

encoding. Note that the syntax for assigning a value to a constant uses the := assignment operator, rather than the <= operator that is used for signals. When the code is translated, the VHDL compiler replaces the symbolic names *A*, *B*, and *C* with their assigned constant values.

The CASE statement that defines the state diagram is identical to that in Figure 6.33 with one exception. VHDL requires that the CASE statement for *y_present* include a WHEN clause for all possible values of *y_present*. In Figure 6.33 *y_present* can have only the three values *A*, *B*, and *C* because it has the State_type type. But since *y_present* is a STD_LOGIC_ VECTOR signal in Figure 6.35, we must provide a WHEN OTHERS clause, as shown. In practice, the machine should never enter the unused state, which corresponds to *y_present* = 10. As we said earlier, there is a slight possibility that this could occur due to erroneous behavior of the circuit. As a pragmatic choice, we have specified that the FSM should change back to the reset state if such an error occurs.

6.4.7 SPECIFICATION OF MEALY FSMS USING VHDL

A Mealy-type FSM can be specified in a similar manner as a Moore-type FSM. Figure 6.36 gives complete VHDL code for the FSM in Figure 6.23. The state transitions are described in the same way as in our original VHDL example in Figure 6.29. The signal *y* represents the state flip-flops, and State_type specifies that *y* can have the values *A* and *B*. Compared to the code in Figure 6.29, the major difference in the case of a Mealy-type FSM is the way in which the code for the output is written. In Figure 6.36 the output *z* is defined using a CASE statement. It states that when the FSM is in state *A*, *z* should be 0, but when in state *B*, *z* should take the value of *w*. This CASE statement properly describes the logic needed for *z*, but it may not be obvious why we have used a second CASE statement in the code, rather than specifying the value of *z* inside the CASE statement that defines the state transitions. The reason is that the CASE statement for the state transitions is nested inside the ELSIF clause that depends on the clock edge. Hence if we placed the code for *z* inside this CASE statement, then the value of *z* could change only as a result of a clock edge. This does not meet the requirements of the Mealy-type FSM, because the value of *z* must depend not only on the state of the machine but also on the input *w*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mealy IS
    PORT ( Clock, Resetn, w  : IN   STD_LOGIC ;
           z                 : OUT  STD_LOGIC ) ;
END mealy ;

ARCHITECTURE Behavior OF mealy IS
    TYPE State_type IS (A, B) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    PROCESS ( y, w )
    BEGIN
        CASE y IS
            WHEN A =>
                z <= '0' ;
            WHEN B =>
                z <= w ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 6.36 VHDL code for the Mealy machine in Figure 6.23.

Implementing the FSM specified in Figure 6.36 in a chip yields the same equations as we derived manually in Section 6.3. Simulation results for the synthesized circuit appear in Figure 6.37. The input waveform for *w* is the same as the one we used for the Moore-type machine in Figure 6.32. Our Mealy-type machine behaves correctly, with *z* becoming 1 just after the start of the second consecutive clock cycle in which *w* is 1.

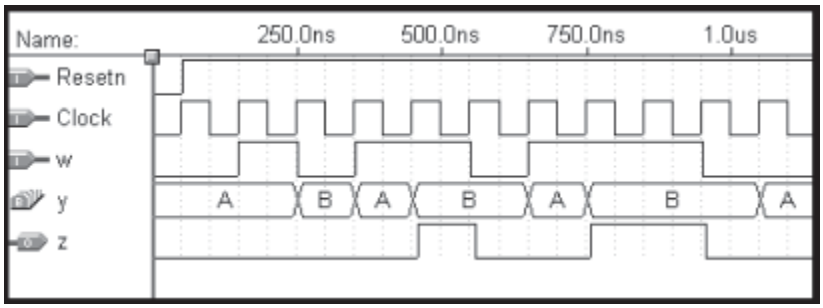


Figure 6.37 Simulation results for the Mealy machine.

In the simulation results we have given in this section, all changes in the input *w* occur immediately following a positive clock edge. This is based on the assumption stated in Section 6.1.5 that in a real circuit *w* would be synchronized with respect to the clock that controls the FSM. In Figure 6.38 we illustrate a problem that may arise if *w* does not meet this specification. In this case we have assumed that the changes in *w* take place at the negative edge of the clock, rather than at the positive edge when the FSM changes its state. The first pulse on the *w* input is 100 ns long. This should not cause the output *z* to become equal to 1. But the circuit does not behave in this manner. After the signal *w* becomes equal to 1, the first positive edge of the clock causes the FSM to change from state *A* to state *B*. As soon as the circuit reaches the state *B*, the *w* input is still equal to 1 for another 50 ns, which causes *z* to go to 1. When *w* returns to 0, the *z* signal does likewise. Thus an erroneous 50-ns pulse is generated on the output *z*.

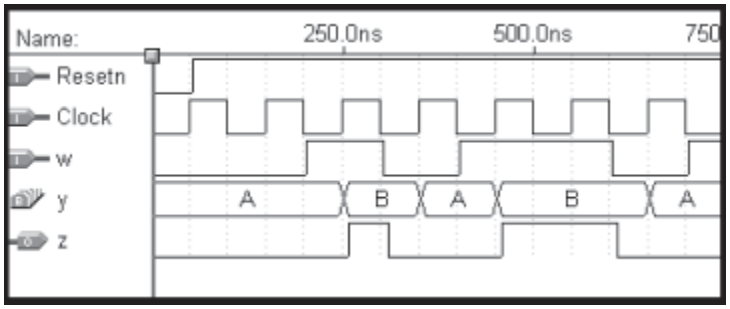


Figure 6.38 Potential problem with asynchronous inputs to a Mealy FSM.

We should pursue the consequences of this problem a little further. If *z* is used to drive another circuit that is not controlled by the same clock, then the extraneous pulse is likely to cause big problems. But if *z* is used as an input to a circuit (perhaps another FSM) that is controlled by the same

clock, then the 50-ns pulse will be ignored by this circuit if $z = 0$ before the next positive edge of the clock (accounting for the setup time).

6.5 Serial Adder Example

We will now present another simple example that illustrates the complete design process. In Chapter 3 we discussed the addition of binary numbers in detail. We explained several schemes that can be used to add two n-bit numbers in parallel, ranging from carry-ripple to carry-lookahead adders. In these schemes the speed of the adder unit is an important design parameter. Fast adders are more complex and thus more expensive. If speed is not of great importance, then a cost-effective option is to use a *serial adder*, in which bits are added a pair at a time.

6.5.1 Mealy-Type FSM for Serial Adder

Let $A = a_{n-1}a_{n-2} \cdots a_0$ and $B = b_{n-1}b_{n-2} \cdots b_0$ be two unsigned numbers that have to be added to produce $Sum = s_{n-1}s_{n-2} \cdots s_0$. Our task is to design a circuit that will perform serial addition, dealing with a pair of bits in one clock cycle. The process starts by adding bits a_0 and b_0. In the next clock cycle, bits a_1 and b_1 are added, including a possible carry from the bit-position 0, and so on. Figure 6.39 shows a block diagram of a possible implementation. It includes three shift registers that are used to hold A, B, and Sum as the computation proceeds. Assuming that the input shift registers have parallel-load capability, as depicted in Figure 5.18, the addition task begins by loading the values of A and B into these registers. Then in each clock cycle, a pair of bits is added by the adder FSM, and at the end of the cycle the resulting sum bit is shifted into the Sum register. We will use positive-edge-triggered flip-flops, in which case all changes take place soon after the positive edge of the clock, depending on the propagation delays within the various flip-flops. At this time the contents of all three shift registers are shifted to the right; this shifts the existing sum bit into Sum, and it presents the next pair of input bits a_i and b_i to the adder FSM.

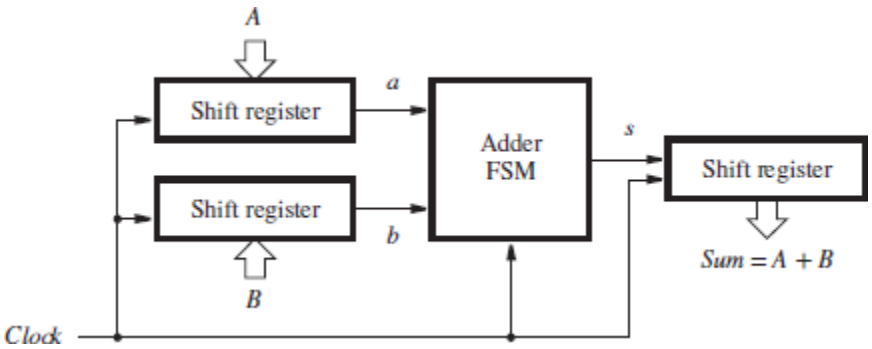


Figure 6.39 Block diagram for the serial adder.

Now we are ready to design the required FSM. This cannot be a combinational circuit because different actions will have to be taken, depending on the value of the carry from the previous bit position. Hence two states are needed: let *G* and *H* denote the states where the carry-in values are 0 and 1, respectively. Figure 6.40 gives a suitable state diagram, defined as a Mealy model. The output value, *s*, depends on both the state and the present value of the inputs *a* and *b*. Each transition is labeled using the notation *ab/s*, which indicates the value of *s* for a given valuation *ab*. In state *G* the input valuation 00 will produce $s = 0$, and the FSM will remain in the same state. For input valuations 01 and 10, the output will be $s = 1$, and the FSM will remain in *G*. But for 11, $s = 0$ is generated, and the machine moves to state *H*. In state *H* valuations 01 and 10 cause $s = 0$, while 11 causes $s = 1$. In all three of these cases, the machine remains in *H*. However, when the valuation 00 occurs, the output of 1 is produced and a change into state *G* takes place.

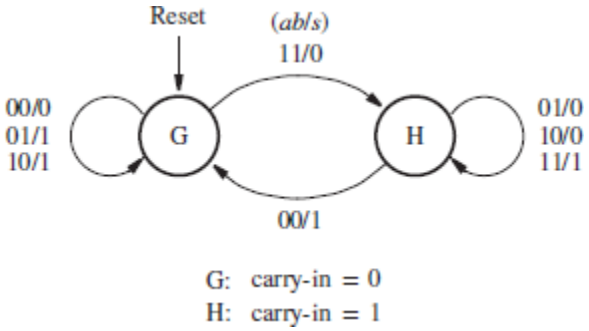


Figure 6.40 State diagram for the serial adder FSM.

The corresponding state table is presented in Figure 6.41. A single flip-flop is needed to represent the two states. The state assignment can be done

as indicated in Figure 6.42.

Present state	Next state				Output			
	$ab = 00$	01	10	11	00	01	10	11
y	Y				s			
0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	0	1

Present state	Next state				Output s			
	$ab = 00$	01	10	11	00	01	10	11
G	G	G	G	H	0	1	1	0
H	G	H	H	H	1	0	0	1

Figure 6.41 State table for the serial adder FSM.

Figure 6.42 State-assigned table for Figure 6.41.

This assignment leads to the following next-state and output equations

$$Y = ab + ay + by$$
$$s = a \oplus b \oplus y$$

Comparing these expressions with those for the full-adder in Section 3.2, it is obvious that y is the carry-in, Y is the carry-out, and s is the sum of the full-adder. Therefore, the adder FSM box in Figure 6.39 consists of the circuit shown in Figure 6.43. The flip-flop can be cleared by the active-high *Reset* signal at the start of the addition operation.

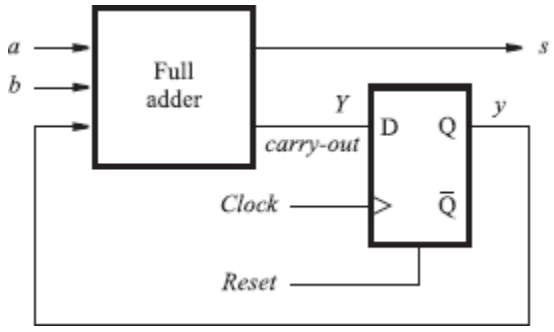


Figure 6.43 Circuit for the adder FSM in Figure 6.39.

The serial adder is a simple circuit that can be used to add numbers of any length. The structure in Figure 6.39 is limited in length only by the size of the shift registers.

6.5.2 MOORE-TYPE FSM FOR SERIAL ADDER

In the preceding example we saw that a Mealy-type FSM nicely meets the requirement for implementing the serial adder. Now we will try to achieve the same objective using a Moore-type FSM. A good starting point is the state diagram in Figure 6.40. In a Moore-type FSM, the output must depend only on the state of the machine. Since in both states, G and H, it is possible to produce two different outputs depending on the valuations of the inputs a and b, a Moore-type FSM will need more than two states. We can derive a suitable state diagram by splitting both G and H into two states. Instead of G, we will use G_0 and G_1 to denote the fact that the carry is 0 and that the sum is either 0 or 1, respectively. Similarly, instead of H, we will use H_0 and H_1. Then the information in Figure 6.40 can be mapped into the Moore-type state diagram in Figure 6.44 in a straightforward manner.

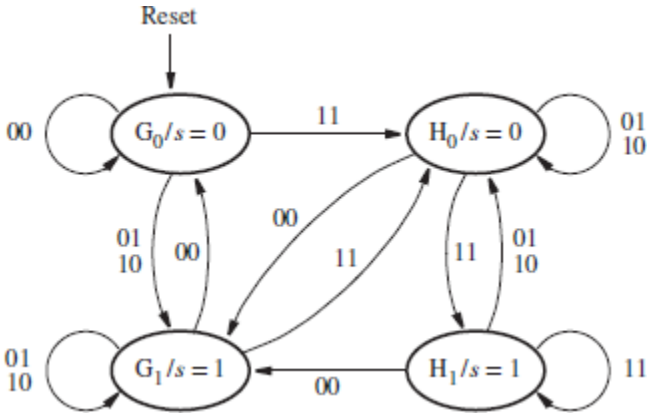


Figure 6.44 State diagram for the Moore-type serial adder FSM.

The corresponding state table is given in Figure 6.45 and the state-assigned table in Figure 6.46. The next-state and output expressions are

Present state	Next state				Output
	$ab = 00$	01	10	11	s
G_0	G_0	G_1	G_1	H_0	0
G_1	G_0	G_1	G_1	H_0	1
H_0	G_1	H_0	H_0	H_1	0
H_1	G_1	H_0	H_0	H_1	1

Figure 6.45 State table for the Moore-type serial adder FSM.

Present state y_2y_1	Next state				Output
	$ab = 00$	01	10	11	s
	Y_2Y_1				
0 0	0 0	0 1	0 1	1 0	0
0 1	0 0	0 1	0 1	1 0	1
1 0	0 1	1 0	1 0	1 1	0
1 1	0 1	1 0	1 0	1 1	1

Figure 6.46 State-assigned table for Figure 6.45.

$$Y_1 = a \oplus b \oplus y_2$$
$$Y_2 = ab + ay_2 + by_2$$
$$s = y_1$$

The expressions for Y_1 and Y_2 correspond to the sum and carry-out expressions in the full-adder circuit. The FSM is implemented as shown in Figure 6.47. It is interesting to observe that this circuit is very similar to the circuit in Figure 6.43. The only difference is that in the Moore-type circuit, the output signal, s, is passed through an extra flip-flop and thus delayed by one clock cycle with respect to the Mealy-type sequential circuit. Recall that we observed the same difference in our previous example, as depicted in Figures 6.26 and 6.27.

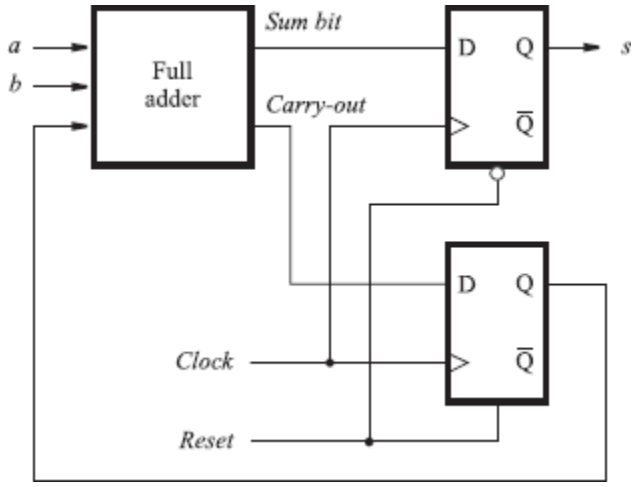


Figure 6.47 Circuit for the Moore-type serial adder FSM.

A key difference between the Mealy and Moore types of FSMs is that in the former a change in inputs reflects itself immediately in the outputs, while in the latter the outputs do not change until the change in inputs forces the machine into a new state, which takes place one clock cycle later. We encourage the reader to draw the timing diagrams for the circuits in Figures 6.43 and 6.47, which will exemplify further this key difference between the two types of FSMs.

6.5.3 VHDL CODE FOR THE SERIAL ADDER

The serial adder can be described in VHDL by writing code for the shift registers and the adder FSM. We will first design the shift register and then use it as a subcircuit in the serial adder.

Shift Register Subcircuit

Figure 5.47 gives VHDL code for an *n*-bit shift register. In the serial adder it is beneficial to have the ability to prevent the shift register contents from changing when an active clock edge occurs. Figure 6.48 gives the code for a shift register named *shiftrne*, which has an enable input, *E*. When $E = 1$, the contents of the register are shifted left-to-right on the positive edge of the clock. Setting $E = 0$ prevents the contents of the shift register from changing.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

-- left-to-right shift register with parallel load and enable
ENTITY shiftrne IS
    GENERIC ( N : INTEGER := 4 ) ;
    PORT ( R       : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           L, E, w : IN     STD_LOGIC ;
           Clock   : IN     STD_LOGIC ;
           Q       : BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END shiftrne ;

ARCHITECTURE Behavior OF shiftrne IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF L = '1' THEN
            Q <= R ;
        ELSIF E = '1' THEN
            Genbits: FOR i IN 0 TO N-2 LOOP
                Q(i) <= Q(i+1) ;
            END LOOP ;
            Q(N-1) <= w ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.48 Code for a left-to-right shift register with an enable input.

Complete Code

The code for the serial adder is shown in Figure 6.49. It instantiates three shift registers for the inputs *A* and *B* and the output *Sum*. The shift registers are loaded with parallel data when the circuit is reset. The state diagram for the adder FSM is described by a single process, using the style of code in Figure 6.29. In addition to the components of the serial adder shown in Figure 6.39, the VHDL code includes a down-counter to determine when the adder should be halted because all *n* bits of the required sum are present in the output shift register. When the circuit is reset, the counter is loaded with the number of bits in the serial adder, *n*. The counter counts down to 0, and then stops and disables further changes in the output shift register.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY serial IS
4       GENERIC ( length : INTEGER := 8 ) ;
5       PORT ( Clock  : IN      STD_LOGIC ;
6              Reset  : IN      STD_LOGIC ;
7              A, B   : IN      STD_LOGIC_VECTOR(length-1 DOWNTO 0) ;
8              Sum    : BUFFER  STD_LOGIC_VECTOR(length-1 DOWNTO 0) );
9   END serial ;

10  ARCHITECTURE Behavior OF serial IS
11      COMPONENT shiftrne
12          GENERIC ( N : INTEGER := 4 ) ;
13          PORT( R       : IN      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
14                L, E, w : IN      STD_LOGIC ;
15                Clock   : IN      STD_LOGIC ;
16                Q       : BUFFER  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
17      END COMPONENT ;

18      SIGNAL QA, QB, Null_in : STD_LOGIC_VECTOR(length-1 DOWNTO 0) ;
19      SIGNAL s, Low, High, Run : STD_LOGIC ;
20      SIGNAL Count : INTEGER RANGE 0 TO length ;
21      TYPE State_type IS (G, H) ;
22      SIGNAL y : State_type ;

        ...continued in Part b
```

Figure 6.49 VHDL code for the serial adder (Part *a*).

```
23 BEGIN
24     Low <= '0' ; High <= '1' ;
25     ShiftA: shiftrne GENERIC MAP (N => length)
26         PORT MAP ( A, Reset, High, Low, Clock, QA ) ;
27     ShiftB: shiftrne GENERIC MAP (N => length)
28         PORT MAP ( B, Reset, High, Low, Clock, QB ) ;
29     AdderFSM: PROCESS ( Reset, Clock )
30     BEGIN
31         IF Reset = '1' THEN
32             y <= G ;
33         ELSIF Clock'EVENT AND Clock = '1' THEN
34             CASE y IS
35                 WHEN G =>
36                     IF QA(0) = '1' AND QB(0) = '1' THEN y <= H ;
37                     ELSE y <= G ;
38                     END IF ;
39                 WHEN H =>
40                     IF QA(0) = '0' AND QB(0) = '0' THEN y <= G ;
41                     ELSE y <= H ;
42                     END IF ;
43             END CASE ;
44         END IF ;
45     END PROCESS AdderFSM ;

46     WITH y SELECT
47         s <= QA(0) XOR QB(0) WHEN G,
48              NOT ( QA(0) XOR QB(0) ) WHEN H ;
49     Null_in <= (OTHERS => '0') ;
50     ShiftSum: shiftrne GENERIC MAP ( N => length )
51                 PORT MAP ( Null_in, Reset, Run, s, Clock, Sum ) ;
52     Stop: PROCESS
53     BEGIN
54         WAIT UNTIL (Clock'EVENT AND Clock = '1') ;
55         IF Reset = '1' THEN
56             Count <= length ;
57         ELSIF Run = '1' THEN
58             Count <= Count - 1 ;
59         END IF ;
60     END PROCESS ;
61     Run <= '0' WHEN Count = 0 ELSE '1' ; -- stops counter and ShiftSum
62 END Behavior ;
```

Figure 6.49 VHDL code for the serial adder (Part *b*).

The lines of code in Figure 6.49 are numbered on the left for reference. The GENERIC parameter *length* sets the number of bits in the serial adder. Since the value of *length* is equal to 8, the code represents a serial adder for eight-bit numbers. By changing the value of *length*, the same code can be used to synthesize a serial adder circuit for any number of bits.

Lines 18 to 22 define several signals used in the code. The signals Q*A* and Q*B* correspond to the parallel outputs of the shift registers with inputs *A* and *B* in Figure 6.39. The signal named *s* represents the output of the adder FSM. The other signals will be described along with the lines of code where they are used.

In Figure 6.39 the shift registers for inputs *A* and *B* do not use a serial input or an enable input. However, the *shiftrne* component, which is used for all three shift registers, includes these ports and so signals must be connected to them. The enable input for the two shift registers can be connected to logic value 1. The value shifted into the serial input does not matter, so it can be connected to either 1 or 0. In lines 26 and 28, the enable input is connected to the signal named *High*, which is set to 1, and the serial inputs are tied to the signal *Low*, which is 0. These signals are needed because the VHDL syntax does not allow the constants 0 or 1 to be attached to the ports of a component. The *N* parameter for each shift register is set to *length* using GENERIC MAP. If the GENERIC MAP were not provided, then the default value of $N = 4$ given in the code in Figure 6.48 would be used. The shift registers are loaded in parallel by the *Reset* signal. We have chosen to use an active-high reset signal for the circuit.

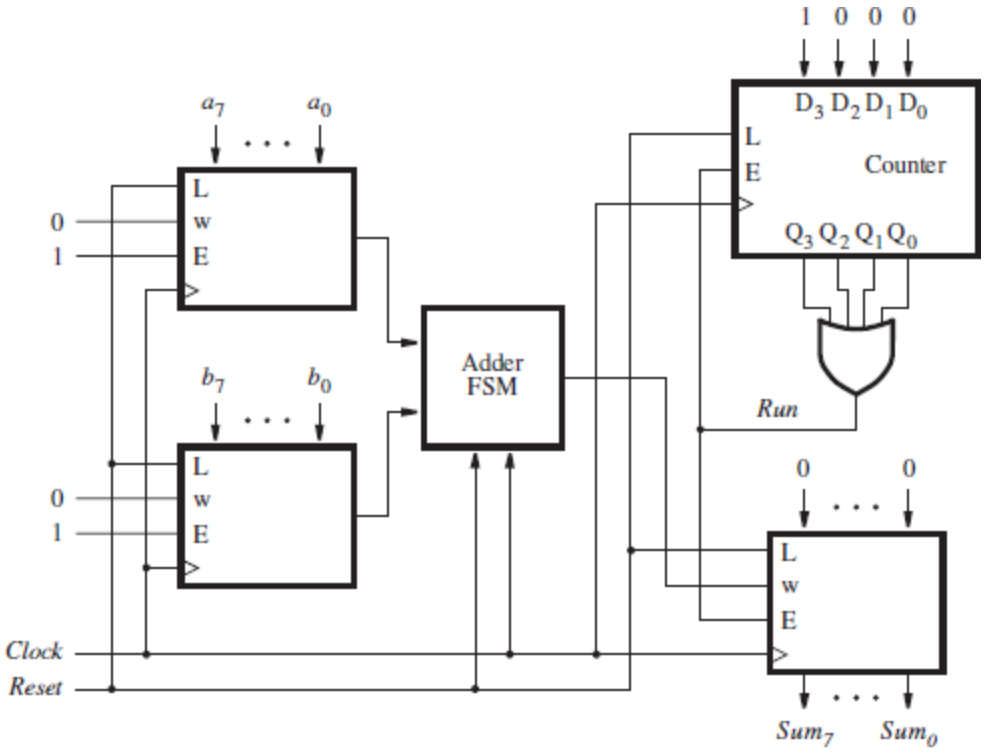
The adder FSM is specified in lines 29 to 45, which describes the state transitions in Figure 6.41. Lines 46 to 48 define the output, *s*, of the adder FSM. This statement results from observing in Figure 6.41 that when the FSM is in state *G*, the sum is $s = a \oplus b$, and when in state *H*, the sum is $s = \overline{a \oplus b}$.

The output shift register does not need a parallel data input. But since the *shiftrne* component has this port, a signal must be connected to it. The signal named *Null_in* is used for this purpose. Line 49 sets *Null_in*, which is a STD_LOGIC_VECTOR signal, to all 0s. The number of bits in this signal is defined by the *length* parameter. Hence we cannot use the normal VHDL syntax, namely, a string of 0s inside double quotes, to set all of its bits to 0. A solution to this problem is to use the syntax (OTHERS => ‘0’), which we explained in the discussion regarding Figure 5.42. The enable input for the shift register is named *Run*. It is derived from the outputs of the down-counter described in lines 52 to 60. When *Reset* =1, *Count* is initialized to the value of *length*. Then as long as *Run* =1, *Count* is decremented in each clock cycle. In line 61 *Run* is set to 0 when *Count* is equal to 0. Note that no

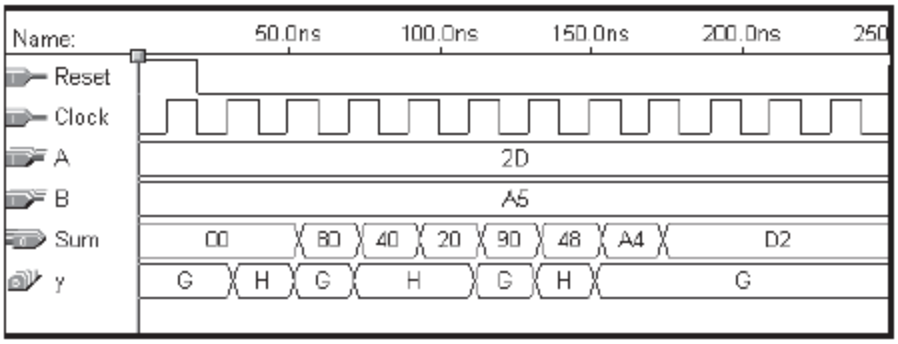
quotes are used in the condition *Count* = 0, because the 0 without quotes has the integer type.

Synthesis and Simulation of the VHDL Code

The results of synthesizing a circuit from the code in Figure 6.49 are illustrated in Figure 6.50*a*. The outputs of the counter are ORed to provide the *Run* signal, which enables clocking of both the output shift register and the counter. A sample of a simulation for the circuit is shown in Figure 6.50*b*. The circuit is first reset, resulting in the values of *A* and *B* being loaded into the input shift registers, and the value of *length* (8) loaded into the down-counter. After each clock cycle one pair of bits of the input numbers is added by the adder FSM, and the sum bit is shifted into the output shift register. After eight clock cycles the output shift register contains the correct sum, and shifting is halted by the *Run* signal becoming equal to 0.



(a) Circuit



(b) Simulation Results

Figure 6.50 Synthesized serial adder.

6.6 State Minimization

Our introductory examples of finite state machines were so simple that it was easy to see that the number of states that we used was the minimum

possible to perform the required function. When a designer has to design a more complex FSM, it is likely that the initial attempt will result in a machine that has more states than is actually required. Minimizing the number of states is of interest because fewer flips-flops may be needed to represent the states and the complexity of the combinational circuit needed in the FSM may be reduced.

If the number of states in an FSM can be reduced, then some states in the original design must be equivalent to other states in their contribution to the overall behavior of the FSM. We can express this more formally in the following definition.

Definition 6.1 – *Two states S_i and S_j are said to be equivalent if and only if for every possible input sequence, the same output sequence will be produced regardless of whether S_i or S_j is the initial state.*

It is possible to define a minimization procedure that searches for any states that are equivalent. Such a procedure is very tedious to perform manually, but it can be automated for use in CAD tools. We will not pursue it here, because of its tediousness. However, to provide some appreciation of the impact of state minimization, we will present an alternative approach, which is much more efficient but not quite as broad in scope.

Instead of trying to show that some states in a given FSM are equivalent, it is often easier to show that some states are definitely **not** equivalent. This idea can be exploited to define a simple minimization procedure.

6.6.1 PARTITIONING MINIMIZATION PROCEDURE

Suppose that a state machine has a single input w. Then if the input signal $w = 0$ is applied to this machine in state S_i and the result is that the machine moves to state S_u, we will say that S_u is a 0-*successor* of S_i. Similarly, if $w = 1$ is applied in the state S_i and it causes the machine to move to state S_v, we will say that S_v is a 1-*successor* of S_i. In general, we will refer to the successors of S_i as its k-successors. When the FSM has only one input, k can be either 0 or 1. But if there are multiple inputs to the FSM, then k represents the set of all possible combinations (valuations) of the inputs.

From Definition 6.1 it follows that if the states S_i and S_j are equivalent, then their corresponding k-successors (for all k) are also equivalent. Using

this fact, we can formulate a minimization procedure that involves considering the states of the machine as a set and then breaking the set into *partitions* that comprise subsets that are definitely not equivalent.

Definition 6.2 – *A partition consists of one or more blocks, where each block comprises a subset of states that may be equivalent, but the states in a given block are definitely not equivalent to the states in other blocks.*

Let us assume initially that all states are equivalent; this forms the initial partition, P_1, in which all states are in the same block. As the next step, we will form the partition P_2 in which the set of states is partitioned into blocks such that the states in each block generate the same output values. Obviously, the states that generate different outputs cannot possibly be equivalent. Then we will continue to form new partitions by testing whether the *k*-successors of the states in each block are contained in one block. Those states whose *k*-successors are in different blocks cannot be in one block. Thus new blocks are formed in each new partition. The process ends when a new partition is the same as the previous partition. Then all states in any one block are equivalent. To illustrate the procedure, consider Example 6.5.

Example 6.5

Figure 6.51 shows a state table for a particular FSM. In an attempt to minimize the number of states, let us apply the partitioning procedure. The initial partition contains all states in a single block

$$P_1 = (ABCDEFG)$$

The next partition separates the states that have different outputs (note that this FSM is of Moore type), which means that the states *A*, *B*, and *D* must be different from the states *C*, *E*, *F*, and *G*. Thus the new partition has two blocks

$$P_2 = (ABD)(CEFG)$$

Now we must consider all 0- and 1-successors of the states in each block. For the block (*ABD*), the 0-successors are (*BDB*), respectively. Since all of

these successor states are in the same block in P_2, we should still assume that the states *A*, *B*, and *D* may be equivalent. The 1-successors for these states are (*CFG*). Since these successors are also in the same block in P_2, we conclude that (*ABD*) should remain in one block of P_3. Next consider the block (*CEFG*). Its 0-successors are (*FFEF*), respectively. They are in the same block in P_2. The 1-successors are (*ECDG*). Since these states are not in the same block in P_2, it means that at least one of the states in the block (*CEFG*) is not equivalent to the others. In particular, the state *F* must be different from the states *C*, *E*, and *G* because its 1-successor is *D*, which is in a different block than *C*, *E*, and *G*. Hence

$$P_3 = (ABD)(CEG)(F)$$

Repeating the process yields the following. The 0-successors of (*ABD*) are (*BDB*), which are in the same block of P_3. The 1-successors are (*CFG*), which are not in the same block. Since *F* is in a different block than *C* and *G*, it follows that the state *B* cannot be equivalent to states *A* and *D*. The 0- and 1-successors of (*CEG*) are (*FFF*) and (*ECG*), respectively. Both of these subsets are accommodated in the blocks of P_3. Therefore

$$P_4 = (AD)(B)(CEG)(F)$$

If we follow the same approach to check the 0- and 1-successors of the blocks (*AD*) and (*CEG*), we find that

$$P_5 = (AD)(B)(CEG)(F)$$

Since $P_5 = P_4$ and no new blocks are generated, it follows that states in each block are equivalent. If the states in some block were not equivalent, then their *k*-successors would have to be in different blocks. Therefore, states *A* and *D* are equivalent, and *C*, *E*, and *G* are equivalent. Since each block can be represented by a single state, only four states are needed to implement the FSM defined by the state table in Figure 6.51. If we let the symbol *A* represent both the states *A* and *D* in the figure and the symbol *C* represent the states *C*, *E*, and *G*, then the state table reduces to the state table in Figure 6.52.

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	B	C	1
B	A	F	1
C	F	C	0
F	C	A	0

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

Figure 6.51 State table for Example 6.5.

Figure 6.52 Minimized state table for Example 6.5.

The effect of the minimization is that we have found a solution that requires only two flip-flops to realize the four states of the minimized state table, instead of needing three flip-flops for the original design. The expectation is that the FSM with fewer states will be simpler to implement, although this is not always the case.

The state minimization concept is based on the fact that two different FSMs may exhibit identical behavior in terms of the outputs produced in response to all possible inputs. Such machines are functionally equivalent, even though they are implemented with circuits that may be vastly different. In general, it is not easy to determine whether or not two arbitrary FSMs are equivalent. Our minimization procedure ensures that a simplified FSM is functionally equivalent to the original one. We encourage the reader to get an intuitive feeling that the FSMs in Figures 6.51 and 6.52 are indeed functionally equivalent by implementing both machines and simulating their behavior using the CAD tools.

Example 6.6

As another example of minimization, we will consider the design of a sequential circuit that could control a vending machine. Suppose that a coin-operated vending machine dispenses candy under the following conditions:

- The machine accepts nickels and dimes.
- It takes 15 cents for a piece of candy to be released from the machine.
- If 20 cents is deposited, the machine will not return the change, but it will credit the buyer with 5 cents and wait for the buyer to make a second purchase.

All electronic signals in the vending machine are synchronized to the positive edge of a clock signal, named *Clock*. The exact frequency of the clock signal is not important for our example, but we will assume a clock period of 100 ns. The vending machine's coin-receptor mechanism generates two signals, $sense_D$ and $sense_N$, which are asserted when a dime or a nickel is detected. Since the coin receptor is a mechanical device and thus very slow compared to an electronic circuit, inserting a coin causes $sense_D$ or $sense_N$ to be set to 1 for a large number of clock cycles. We will assume that the coin receptor also generates two other signals, named *D* and *N*. The *D* signal is set to 1 for one clock cycle after $sense_D$ becomes 1, and *N* is set to 1 for one clock cycle after $sense_N$ becomes 1. The timing relationships between *Clock*, $sense_D$, $sense_N$, *D*, and *N* are illustrated in Figure 6.53*a*. The hash marks on the waveforms indicate that $sense_D$ or $sense_N$ may be 1 for many clock cycles. Also, there may be an arbitrarily long time between the insertion of two consecutive coins. Note that since the coin receptor can accept only one coin at a time, it is not possible to have both *D* and *N* set to 1 at once. Figure 6.53*b* illustrates how the *N* signal may be generated from the $sense_N$ signal.

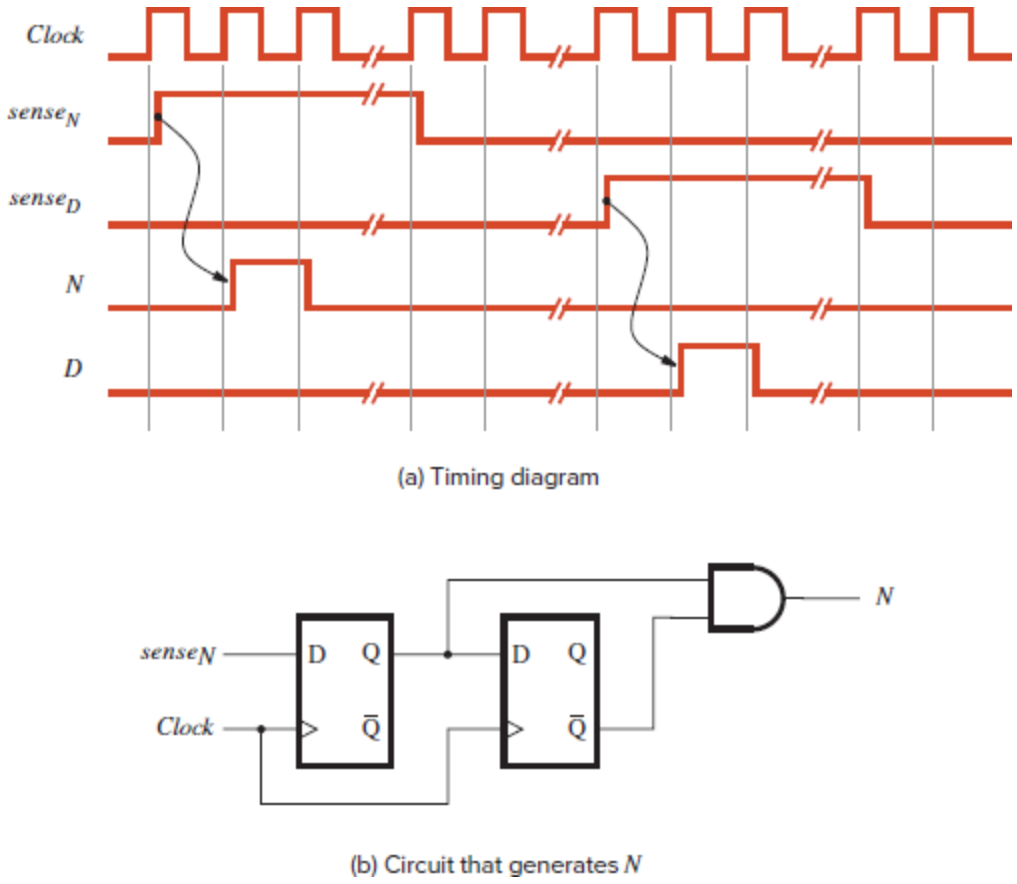


Figure 6.53 Signals for the vending machine.

Based on these assumptions, we can develop an initial state diagram in a fairly straightforward manner, as indicated in Figure 6.54. The inputs to the FSM are D and N, and the starting state is $S1$. As long as $D = N = 0$, the machine remains in state $S1$, which is indicated by the arc labeled $\overline{D} \cdot \overline{N} = 1$. Inserting a dime leads to state $S2$, while inserting a nickel leads to state $S3$. In both cases the deposited amount is less than 15 cents, which is not sufficient to release the candy. This is indicated by the output, z, being equal to 0, as in $S2/0$ and $S3/0$. The machine will remain in state $S2$ or $S3$ until another coin is deposited because $D = N = 0$. In state $S2$ a nickel will cause a transition to $S4$ and a dime to $S5$. In both of these states, sufficient money is deposited to activate the output mechanism that releases the candy; hence the state nodes have the labels $S4/1$ and $S5/1$. In $S4$ the deposited amount is 15 cents, which means that on the next active clock edge the machine should return to the reset state $S1$. The condition $\overline{D} \cdot \overline{N}$ on the arc leaving $S4$ is guaranteed to be true because the machine remains in state $S4$ for only 100 ns, which is far too short a time for a new coin to have been deposited.

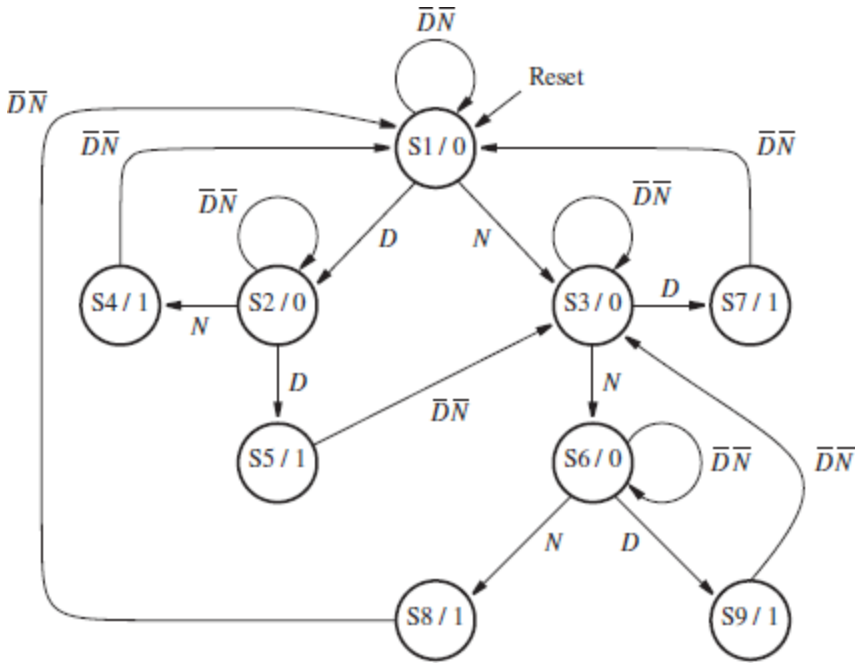


Figure 6.54 State diagram for Example 6.6.

The state *S*5 denotes that an amount of 20 cents has been deposited. The candy is released, and on the next clock edge the FSM makes a transition to state *S*3, which represents a credit of 5 cents. A similar reasoning when the machine is in state *S*3 leads to states *S*6 through *S*9. This completes the state diagram for the desired FSM. A state table version of the same information is given in Figure 6.55.

Present state	Next state				Output
	DN = 00	01	10	11	z
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S6	S7	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1
S6	S6	S8	S9	–	0
S7	S1	–	–	–	1
S8	S1	–	–	–	1
S9	S3	–	–	–	1

Figure 6.55 State table for Example 6.6.

Note that the condition $D = N = 1$ is denoted as don't care in the table. Note also other don't cares in states *S*4, *S*5, *S*7, *S*8, and *S*9. They correspond

to cases where there is no need to check the D and N signals because the machine changes to another state in an amount of time that is too short for a new coin to have been inserted.

Using the minimization procedure, we obtain the following partitions

$$P_1 = (S1, S2, S3, S4, S5, S6, S7, S8, S9)$$
$$P_2 = (S1, S2, S3, S6)(S4, S5, S7, S8, S9)$$
$$P_3 = (S1)(S3)(S2, S6)(S4, S5, S7, S8, S9)$$
$$P_4 = (S1)(S3)(S2, S6)(S4, S7, S8)(S5, S9)$$
$$P_5 = (S1)(S3)(S2, S6)(S4, S7, S8)(S5, S9)$$

The final partition has five blocks. Let $S2$ denote its equivalence to $S6$, let $S4$ denote the same with respect to $S7$ and $S8$, and let $S5$ represent $S9$. This leads to the minimized state table in Figure 6.56. The actual circuit that implements this table can be designed as explained in the previous sections.

Present state	Next state				Output
	$DN = 00$	01	10	11	z
S1	S1	S3	S2	—	0
S2	S2	S4	S5	—	0
S3	S3	S2	S4	—	0
S4	S1	—	—	—	1
S5	S3	—	—	—	1

Figure 6.56 Minimized state table for Example 6.6.

In this example we used a straightforward approach to derive the original state diagram, which we then minimized using the partitioning procedure. Figure 6.57 presents the information in the state table of Figure 6.56 in the form of a state diagram. Looking at this diagram, the reader can probably see that it may have been quite feasible to derive the optimized diagram directly, using the following reasoning. Suppose that the states correspond to the various amounts of money deposited. In particular, the states, $S1$, $S3$, $S2$, $S4$, and $S5$ correspond to the amounts of 0, 5, 10, 15, and 20 cents, respectively. With this interpretation of the states, it is not difficult to derive the transition arcs that define the desired FSM. In practice, the designer can often produce initial designs that do not have a large number of superfluous states.

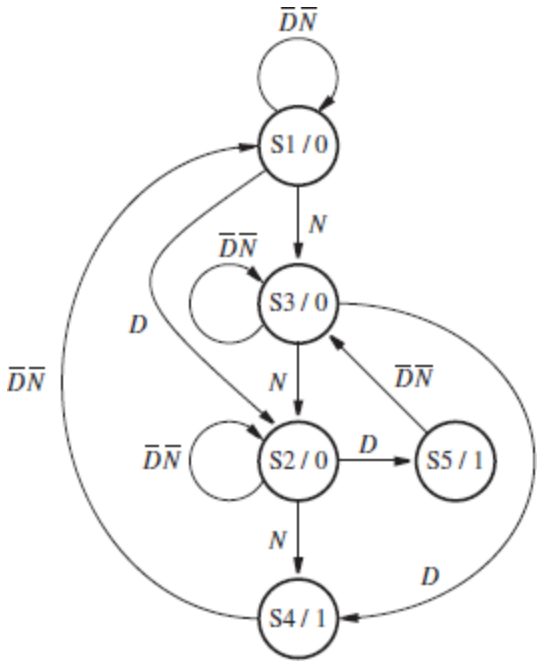


Figure 6.57 Minimized state diagram for Example 6.6.

We have found a solution that requires five states, which is the minimum number of states for a Moore-type FSM that realizes the desired vending control task. From Section 6.3 we know that Mealy-type FSMs may need fewer states than Moore-type machines, although they do not necessarily lead to simpler overall implementations. If we use the Mealy model, we can eliminate states *S*4 and *S*5 in Figure 6.57. The result is shown in Figure 6.58. This version requires only three states, but the output functions become more complicated. The reader is encouraged to compare the complexity of implementations by completing the design steps for the FSMs in Figures 6.57 and 6.58.

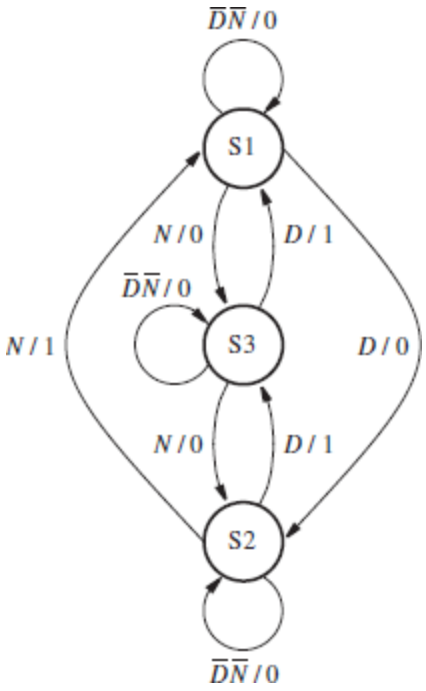


Figure 6.58 Mealy-type FSM for Example 6.6.

6.6.2 INCOMPLETELY SPECIFIED FSMS

The partitioning scheme for minimization of states works well when all entries in the state table are specified. Such is the case for the FSM defined in Figure 6.51. FSMs of this type are said to be *completely specified*. If one or more entries in the state table are not specified, corresponding to don't-care conditions, then the FSM is said to be *incompletely specified*. An example of such an FSM is given in Figure 6.55. As seen in Example 6.6, the partitioning scheme works well for this FSM also. But in general, the partitioning scheme is less useful when incompletely specified FSMs are involved, as illustrated by Example 6.7.

Example 6.7

Consider the FSM in Figure 6.59 which has four unspecified entries, because we have assumed that the input $w = 1$ will not occur when the machine is in states B or G. Accordingly, neither a state transition nor an output value is specified for these two cases. An important difference

between this FSM and the one in Figure 6.55 is that some outputs in this FSM are unspecified, whereas in the other FSM all outputs are specified.

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	D	–	0	–
C	F	E	0	1
D	B	G	0	0
E	F	C	0	1
F	E	D	0	1
G	F	–	0	–

Figure 6.59 Incompletely specified state table for Example 6.7.

The partitioning minimization procedure can be applied to Mealy-type FSMs in the same way as for Moore-type FSMs illustrated in Examples 6.5 and 6.6. Two states are considered equivalent, and are thus placed in the same block of a partition, if their outputs are equal for all corresponding input valuations. To perform the partitioning process, we can assume that the unspecified outputs have a specific value. Not knowing whether these values should be 0 or 1, let us first assume that both unspecified outputs have a value of 0. Then the first two partitions are

$$P_1 = (ABCDEFG)$$
$$P_2 = (ABDG)(CEF)$$

Note that the states *A*, *B*, *D*, and *G* are in the same block because their outputs are equal to 0 for both $w = 0$ and $w = 1$. Also, the states *C*, *E*, and *F* are in one block because they have the same output behavior; they all generate $z = 0$ if $w = 0$, and $z = 1$ if $w = 1$. Continuing the partitioning procedure gives the remaining partitions

$$P_3 = (AB)(D)(G)(CE)(F)$$
$$P_4 = (A)(B)(D)(G)(CE)(F)$$
$$P_5 = P_4$$

The result is an FSM that is specified by six states.

Next consider the alternative of assuming that both unspecified outputs in Figure 6.59 have a value of 1. This would lead to the partitions

$$P_1 = (ABCDEFG)$$
$$P_2 = (AD)(BCEFG)$$
$$P_3 = (AD)(B)(CEFG)$$
$$P_4 = (AD)(B)(CEG)(F)$$
$$P_5 = P_4$$

This solution involves four states. Evidently, the choice of values assigned to unspecified outputs is of considerable importance.

We will not pursue the issue of state minimization of incompletely specified FSMs any further. As we already mentioned, it is possible to develop a minimization technique that searches for equivalent states based directly on Definition 6.1.

Finally, it is important to mention that reducing the number of states in a given FSM will not necessarily lead to a simpler implementation. Interestingly, the effect of state assignment, discussed in Section 6.2, may have a greater influence on the simplicity of implementation than does the state minimization. In a modern design environment, the designer relies on the CAD tools to implement state machines efficiently

6.7 Design of a Counter Using the Sequential Circuit Approach

In this section we discuss the design of a counter circuit as a finite state machine. From Chapter 5 we already know that counters can be realized as cascaded stages of flip-flops and some gating logic, where each stage divides the number of incoming pulses by two. To keep our example simple, we choose a counter of small size but also show how the design can be extended to larger sizes. The specification for the counter is

- The counting sequence is 0, 1, 2, …, 6, 7, 0, 1, …
- There exists an input signal w. The value of this signal is considered during each clock cycle. If $w = 0$, the present count remains the same; if $w = 1$, the count is incremented.

The counter can be designed as a synchronous sequential circuit using the design techniques introduced in the previous sections.

6.7.1 STATE DIAGRAM AND STATE TABLE FOR A MODULO-8 COUNTER

Figure 6.60 gives a state diagram for the desired counter. There is a state associated with each count. In the diagram state *A* corresponds to count 0, state *B* to count 1, and so on. We show the transitions between the states needed to implement the counting sequence. Note that the output signals are specified as depending only on the state of the counter at a given time, which is the Moore model of sequential circuits.

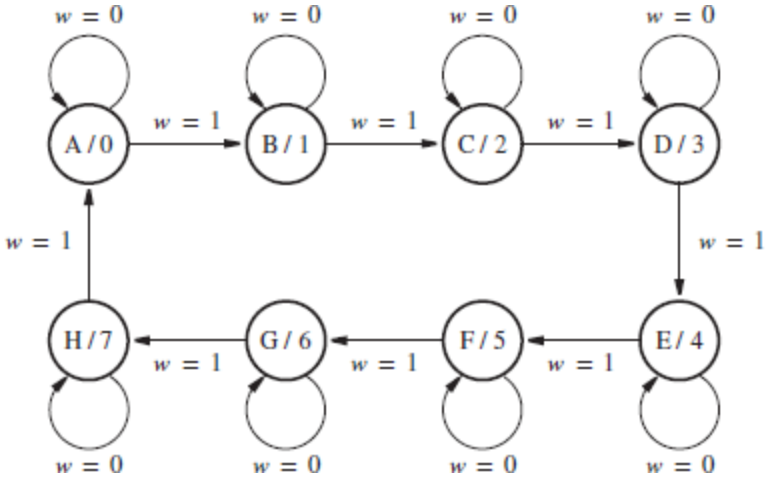


Figure 6.60 State diagram for the counter.

The state diagram may be represented in the state-table form as shown in Figure 6.61.

Present state	Next state		Output
	$w = 0$	$w = 1$	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3
E	E	F	4
F	F	G	5
G	G	H	6
H	H	A	7

Figure 6.61 State table for the counter.

6.7.2 STATE ASSIGNMENT

Three state variables are needed to represent the eight states. Let these variables, denoting the present state, be called y_2, y_1, and y_0. Let Y_2, Y_1, and Y_0 denote the corresponding next-state functions. The most convenient (and simplest) state assignment is to encode each state with the binary number that the counter should give as output in that state. Then the required output signals will be the same as the signals that represent the state variables. This leads to the state-assigned table in Figure 6.62.

	Present state $y_2y_1y_0$	Next state		Count $z_2z_1z_0$
		$w = 0$	$w = 1$	
		$Y_2Y_1Y_0$	$Y_2Y_1Y_0$	
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	101	100
F	101	101	110	101
G	110	110	111	110
H	111	111	000	111

Figure 6.62 State-assigned table for the counter.

The final step in the design is to choose the type of flip-flops and derive the expressions that control the flip-flop inputs. The most straightforward choice is to use D-type flip-flops. We pursue this approach first. Then we show the alternative of using JK-type flip-flops.

6.7.3 IMPLEMENTATION USING D-TYPE FLIP-FLOPS

When using D-type flip-flops to realize the finite state machine, each next-state function, Y_i, is connected to the D input of the flip-flop that implements the state variable y_i. The next-state functions are derived from the information in Figure 6.62. Using Karnaugh maps in Figure 6.63, we obtain the following implementation

$$D_0 = Y_0 = \overline{w}y_0 + w\overline{y}_0$$
$$D_1 = Y_1 = \overline{w}y_1 + y_1\overline{y}_0 + wy_0\overline{y}_1$$
$$D_2 = Y_2 = \overline{w}y_2 + \overline{y}_0y_2 + \overline{y}_1y_2 + wy_0y_1\overline{y}_2$$

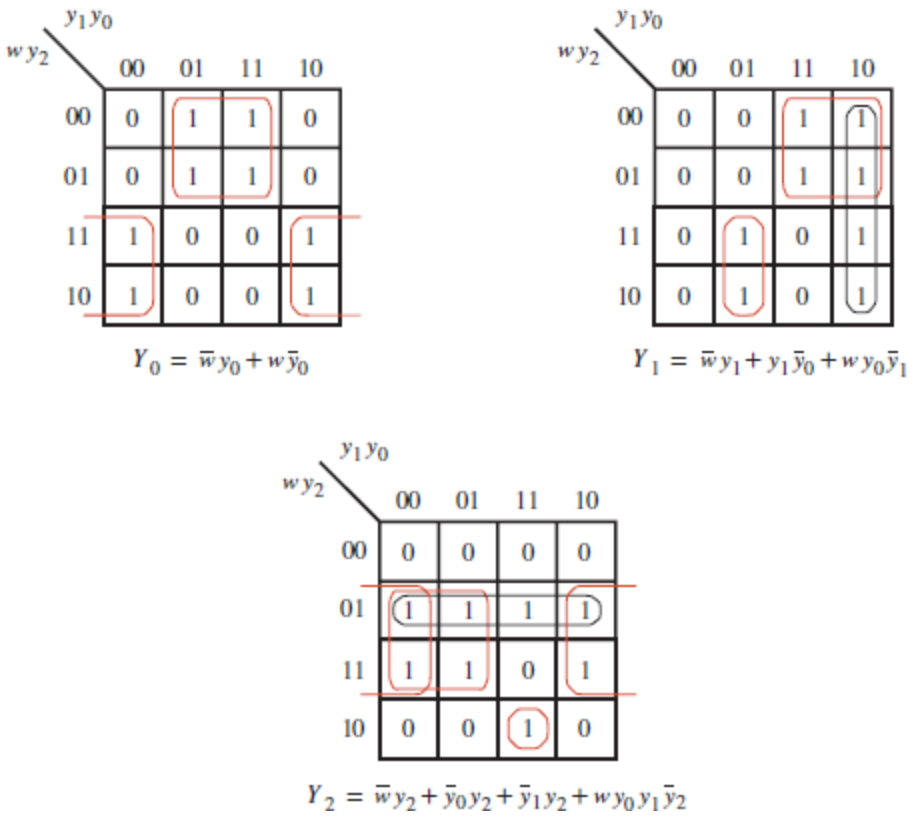


Figure 6.63 Karnaugh maps for D flip-flops for the counter.

The resulting circuit is given in Figure 6.64. It is not obvious how to extend this circuit to implement a larger counter, because no clear pattern is discernible in the expressions for D_0, D_1, and D_2. However, we can rewrite these expressions as follows

$$
\begin{aligned}
D_0 &= \overline{w}y_0 + w\overline{y}_0 \\
&= w \oplus y_0 \\
D_1 &= \overline{w}y_1 + y_1\overline{y}_0 + wy_0\overline{y}_1 \\
&= (\overline{w} + \overline{y}_0)y_1 + wy_0\overline{y}_1 \\
&= \overline{wy_0}y_1 + wy_0\overline{y}_1 \\
&= wy_0 \oplus y_1 \\
D_2 &= \overline{w}y_2 + \overline{y}_0y_2 + \overline{y}_1y_2 + wy_0y_1\overline{y}_2 \\
&= (\overline{w} + \overline{y}_0 + \overline{y}_1)y_2 + wy_0y_1\overline{y}_2 \\
&= \overline{wy_0y_1}y_2 + wy_0y_1\overline{y}_2 \\
&= wy_0y_1 \oplus y_2
\end{aligned}
$$

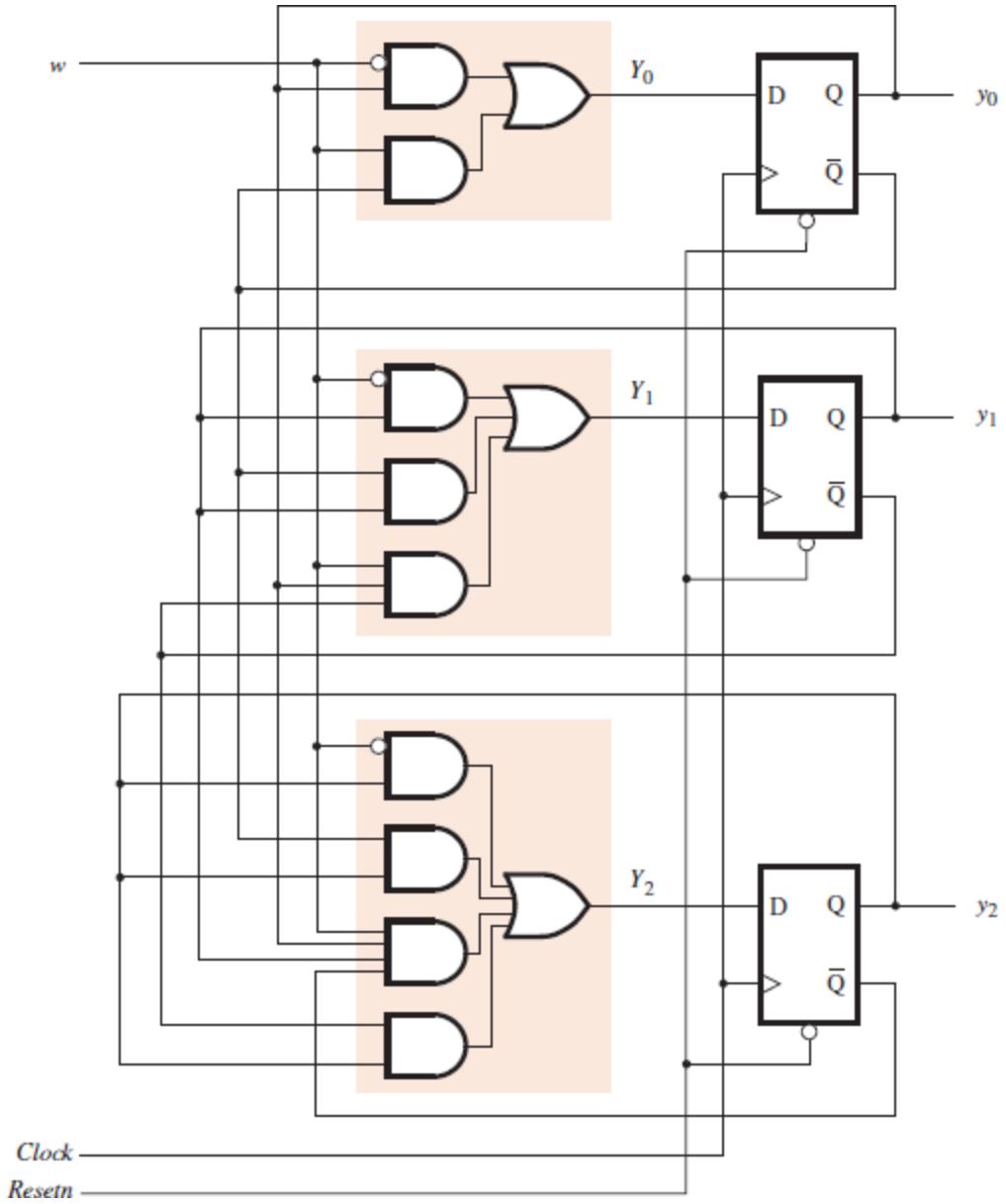


Figure 6.64 Circuit diagram for the counter implemented with D flip-flops.

Then an obvious pattern emerges, which leads to the circuit in Figure 5.23.

6.7.4 IMPLEMENTATION USING JK-TYPE FLIP-FLOPS

JK-type flip-flops provide an attractive alternative. Using these flip-flops to implement the sequential circuit specified in Figure 6.62 requires derivation of *J* and *K* inputs for each flip-flop. The following control is needed:

- If a flip-flop in state 0 is to remain in state 0, then $J = 0$ and $K = d$ (where d means that K can be equal to either 0 or 1).
- If a flip-flop in state 0 is to change to state 1, then $J = 1$ and $K = d$.
- If a flip-flop in state 1 is to remain in state 1, then $J = d$ and $K = 0$.
- If a flip-flop in state 1 is to change to state 0, then $J = d$ and $K = 1$.

Following these guidelines, we can create a truth table that specifies the required values of the J and K inputs for the three flip-flops in our design. Figure 6.65 shows a modified version of the state-assigned table in Figure 6.62, with the J and K input functions included. To see how this table is derived, consider the first row in which the present state is $y_2\, y_1\, y_0 = 000$. If $w = 0$, then the next state is also $Y_2\, Y_1\, Y_0 = 000$. Thus the present value of each flip-flop is 0, and it should remain 0. This implies the control $J = 0$ and $K = d$ for all three flip-flops. Continuing with the first row, if $w = 1$, the next state will be $Y_2\, Y_1\, Y_0 = 001$. Thus flip-flops y_2 and y_1 still remain at 0 and have the control $J = 0$ and $K = d$. However, flip-flop y_0 must change from 0 to 1, which is accomplished with $J = 1$ and $K = d$. The rest of the table is derived in the same manner by considering each present state $y_2\, y_1\, y_0$ and providing the necessary control signals to reach the new state $Y_2\, Y_1\, Y_0$.

	Present state $y_2y_1y_0$	Flip-flop inputs								Count $z_2z_1z_0$
		$w = 0$				$w = 1$				
		$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	
A	000	000	0*d*	0*d*	0*d*	001	0*d*	0*d*	1*d*	000
B	001	001	0*d*	0*d*	*d*0	010	0*d*	1*d*	*d*1	001
C	010	010	0*d*	*d*0	0*d*	011	0*d*	*d*0	1*d*	010
D	011	011	0*d*	*d*0	*d*0	100	1*d*	*d*1	*d*1	011
E	100	100	*d*0	0*d*	0*d*	101	*d*0	0*d*	1*d*	100
F	101	101	*d*0	0*d*	*d*0	110	*d*0	1*d*	*d*1	101
G	110	110	*d*0	*d*0	0*d*	111	*d*0	*d*0	1*d*	110
H	111	111	*d*0	*d*0	*d*0	000	*d*1	*d*1	*d*1	111

Figure 6.65 Excitation table for the counter with JK flip-flops.

A state-assigned table is essentially the state table in which each state is encoded using the state variables. When D flip-flops are used to implement an FSM, the next-state entries in the state-assigned table correspond directly to the signals that must be applied to the D inputs. This is not the case if

some other type of flip-flops is used. A table that gives the state information in the form of the flip-flop inputs that must be "excited" to cause the transitions to the next states is usually called an *excitation table*. The excitation table in Figure 6.65 indicates how JK flip-flops can be used. In many books the term excitation table is used even when D flip-flops are involved, in which case it is synonymous with the state-assigned table.

Once the table in Figure 6.65 has been derived, it provides a truth table with inputs y_2, y_1, y_0, and w, and outputs J_2, K_2, J_1, K_1, J_0, and K_0. We can then derive expressions for these outputs as shown in Figure 6.66. The resulting expressions are

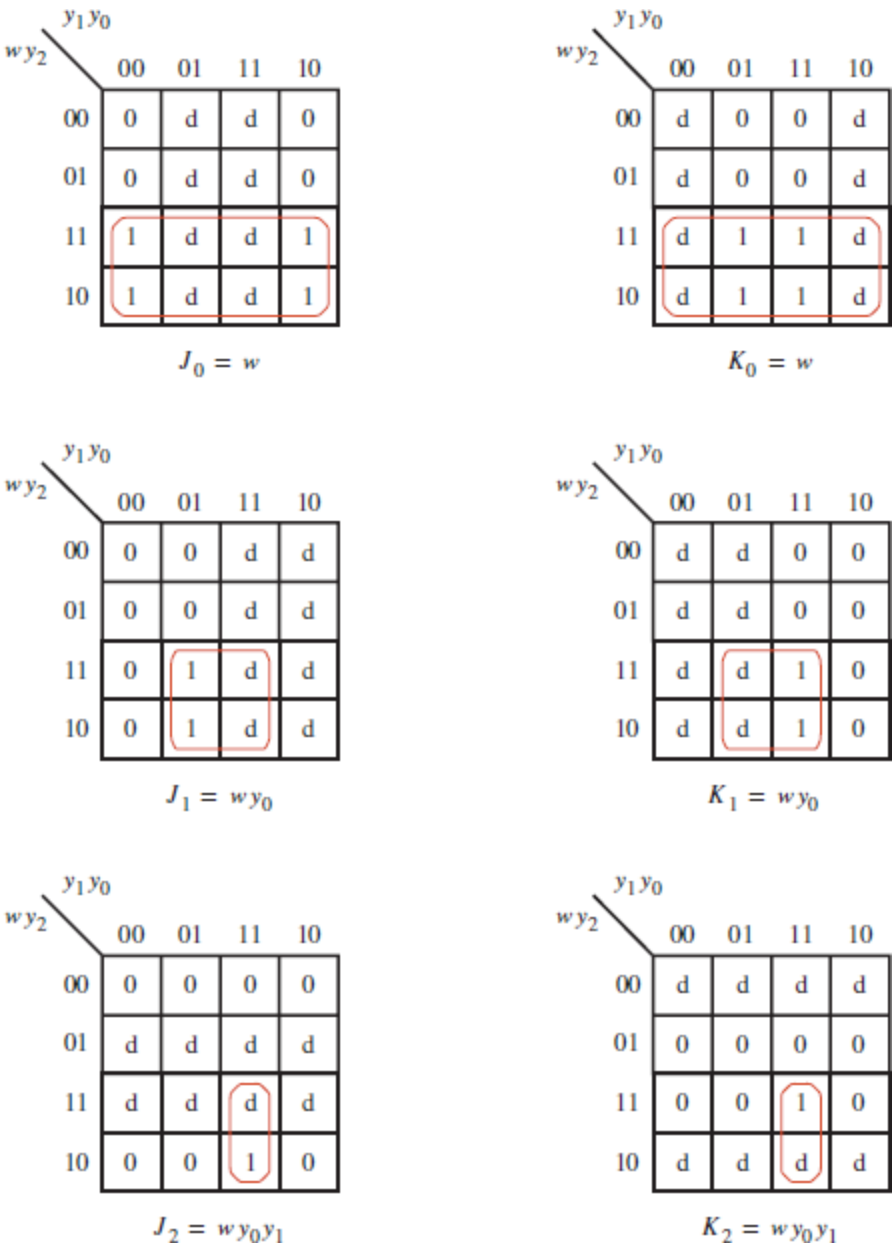


Figure 6.66 Karnaugh maps for JK flip-flops in the counter.

$$J_0 = K_0 = w$$
$$J_1 = K_1 = wy_0$$
$$J_2 = K_2 = wy_0y_1$$

This leads to the circuit shown in Figure 6.67. It is apparent that this design can be extended easily to larger counters. The pattern $J_n = K_n = wy_0y_1 \cdots y_{n-1}$ defines the circuit for each stage in the counter. Note that the size of the

AND gate that implements the product term $y_0\ y_1 \cdots y_{n-1}$ grows with successive stages. A circuit with a more regular structure can be obtained by factoring out the previously needed terms as we progress through the stages of the counter. This gives

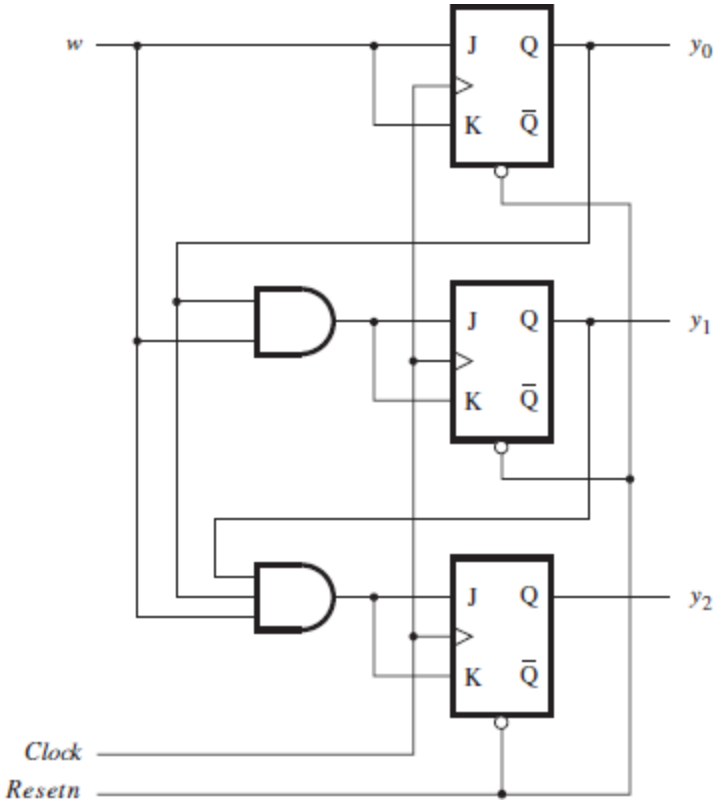


Figure 6.67 Circuit diagram using JK flip-flops.

$$J_2 = K_2 = (wy_0)y_1 \qquad = J_1 y_1$$
$$J_n = K_n = (wy_0 \cdots y_{n-2})y_{n-1} = J_{n-1}y_{n-1}$$

Using the factored form, the counter circuit can be realized as indicated in Figure 6.68. In this circuit all stages (except the first) look the same. Note that this circuit has the same structure as the circuit in Figure 5.22 because connecting the *J* and *K* inputs of a flip-flop together turns the flip-flop into a T flip-flop.

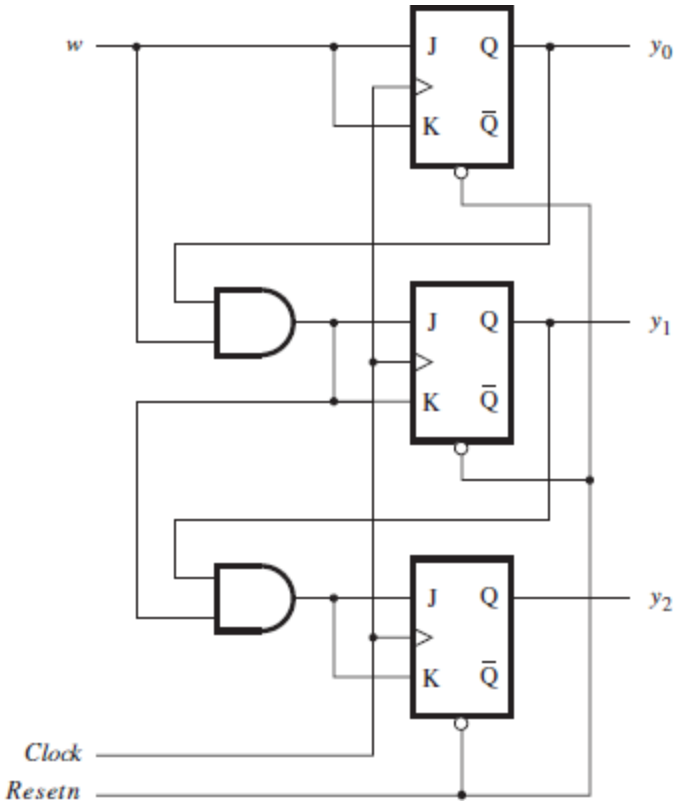


Figure 6.68 Factored form implementation of the counter.

6.7.5 EXAMPLE—A DIFFERENT COUNTER

Having considered the design of an ordinary counter, we will now apply this knowledge to design a slightly different counterlike circuit. Suppose that we wish to derive a three-bit counter that counts the pulses on an input line, *w*. But instead of displaying the count as 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, …, this counter must display the count in the sequence 0, 4, 2, 6, 1, 5, 3, 7, 0, 4, and so on. The count is to be represented directly by the flip-flop values themselves, without using any extra gates. Namely, Count = $Q_2Q_1Q_0$.

Since we wish to count the pulses on the input line *w*, it makes sense to use *w* as the clock input to the flip-flops. Thus the counter circuit should always be enabled, and it should change its state whenever the next pulse on the *w* line appears. The desired counter can be designed in a straightforward manner using the FSM approach. Figures 6.69 and 6.70 give the required state table and a suitable state assignment. Using D flip-flops, we obtain the next-state equations

Present state	Next state	Output $z_2z_1z_0$
A	B	0 0 0
B	C	1 0 0
C	D	0 1 0
D	E	1 1 0
E	F	0 0 1
F	G	1 0 1
G	H	0 1 1
H	A	1 1 1

Figure 6.69 State table for the counterlike example.

Present state $y_2y_1y_0$	Next state $Y_2Y_1Y_0$	Output $z_2z_1z_0$
0 0 0	1 0 0	0 0 0
1 0 0	0 1 0	1 0 0
0 1 0	1 1 0	0 1 0
1 1 0	0 0 1	1 1 0
0 0 1	1 0 1	0 0 1
1 0 1	0 1 1	1 0 1
0 1 1	1 1 1	0 1 1
1 1 1	0 0 0	1 1 1

Figure 6.70 State-assigned table for Figure 6.69.

$$
\begin{aligned}
D_2 &= Y_2 = \overline{y}_2 \\
D_1 &= Y_1 = y_1 \oplus y_2 \\
D_0 &= Y_0 = y_0\overline{y}_1 + y_0\overline{y}_2 + \overline{y}_0y_1y_2 \\
&= y_0(\overline{y}_1 + \overline{y}_2) + \overline{y}_0y_1y_2 \\
&= y_0 \oplus y_1y_2
\end{aligned}
$$

This leads to the circuit in Figure 6.71.

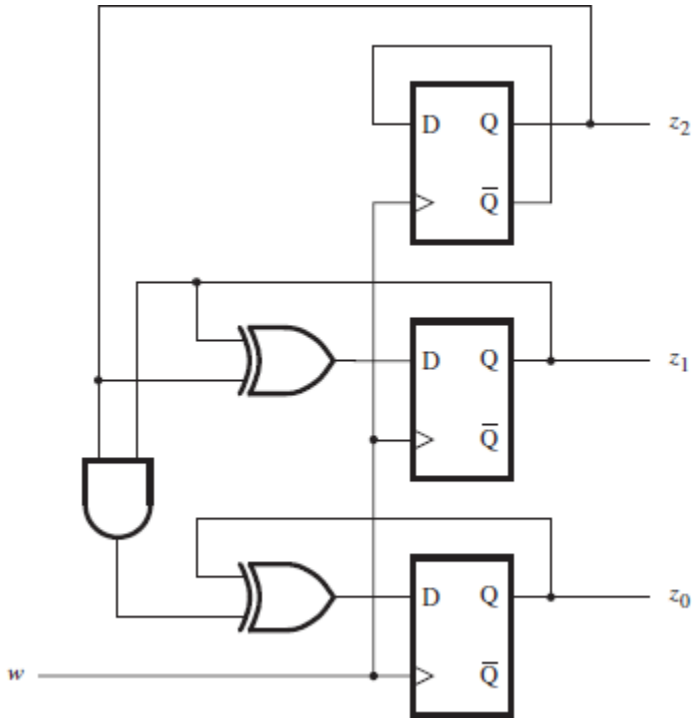


Figure 6.71 Circuit for Figure 6.70.

The reader should compare this circuit with the normal up-counter in Figure 5.23. Take the first three stages of that counter, set the *Enable* input to 1, and let *Clock* = *w*. Then the two circuits are essentially the same with one small difference in the order of bits in the count. In Figure 5.23 the top flip-flop corresponds to the least-significant bit of the count, whereas in Figure 6.71 the top flip-flop corresponds to the most-significant bit of the count. This is not just a coincidence. In Figure 6.70 the required count is defined as $Count = y_2\ y_1\ y_0$. However, if the bit patterns that define the states are viewed in the reverse order and interpreted as binary numbers, such that $Count = y_0\ y_1\ y_2$, then the states *A*, *B*, *C*, …, *H* have the values 0, 1, 2, …, 7. These values are the same as the values that are associated with the normal three-bit up-counter.

6.8 FSM AS AN ARBITER CIRCUIT

In this section we present the design of an FSM that is slightly more complex than the previous examples. The purpose of the machine is to

control access by various devices to a shared resource in a given system. Only one device can use the resource at a time. Assume that all signals in the system can change values only following the positive edge of the clock signal. Each device provides one input to the FSM, called a *request*, and the FSM produces a separate output for each device, called a *grant*. A device indicates its need to use the resource by asserting its request signal. Whenever the shared resource is not already in use, the FSM considers all requests that are active. Based on a priority scheme, it selects one of the requesting devices and asserts its grant signal. When the device is finished using the resource, it deasserts its request signal.

We will assume that there are three devices in the system, called device 1, device 2, and device 3. It is easy to see how the FSM can be extended to handle more devices. The request signals are named r_1, r_2, and r_3, and the grant signals are called g_1, g_2, and g_3. The devices are assigned a priority level such that device 1 has the highest priority, device 2 has the next highest, and device 3 has the lowest priority. Thus if more than one request signal is asserted when the FSM assigns a grant, the grant is given to the requesting device that has the highest priority.

A state diagram for the desired FSM, designed as a Moore-type machine, is depicted in Figure 6.72. Initially, on reset the machine is in the state called *Idle*. No grant signals are asserted, and the shared resource is not in use. There are three other states, called *gnt*1, *gnt*2, and *gnt*3. Each of these states asserts the grant signal for one of the devices.

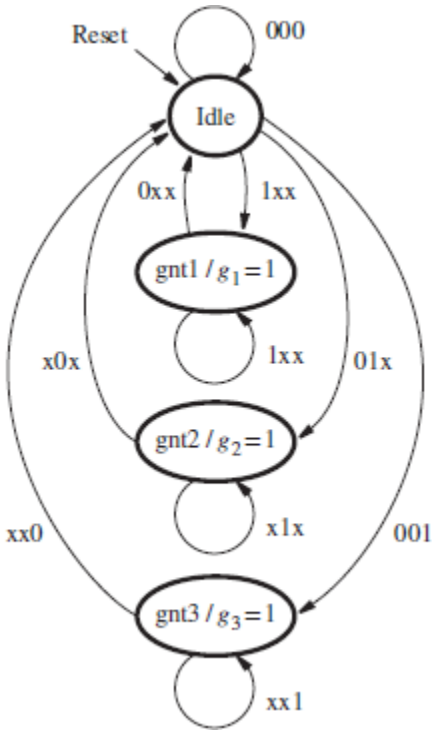


Figure 6.72 State diagram for the arbiter.

The FSM remains in the *Idle* state as long as all of the request signals are 0. In the state diagram the condition $r_1\ r_2\ r_3 = 000$ is indicated by the arc labeled 000. When one or more request signals become 1, the machine moves to one of the grant states, according to the priority scheme. If r_1 is asserted, then device 1 will receive the grant because it has the highest priority. This is indicated by the arc labeled 1xx that leads to state *gnt*1, which sets $g_1 = 1$. The meaning of 1xx is that the request signal r_1 is 1, and the values of signals r_2 and r_3 are irrelevant because of the priority scheme. As before, we use the symbol x to indicate that the value of the corresponding variable can be either 0 or 1. The machine stays in state *gnt*1 as long as r_1 is 1. When $r_1 = 0$, the arc labeled 0xx causes a change on the next positive clock edge back to state *Idle*, and g_1 is deasserted. If other requests are active at this time, then the FSM will change to a new grant state after the next clock edge.

The arc that causes a change to state *gnt*2 is labeled 01x. This label adheres to the priority scheme because it represents the condition that $r_2 =$

1, but $r_1 = 0$. Similarly, the condition for entering state *gnt*3 is given as 001, which indicates that the only request signal asserted is r_3.

The state diagram is repeated in Figure 6.73. The only difference between this diagram and Figure 6.72 is the way in which the arcs are labeled. Figure 6.73 uses a simpler labeling scheme that is more intuitive. For the condition that leads from state *Idle* to state *gnt*1, the arc is labeled as r_1, instead of 1xx. This label means that if $r_1 = 1$, the FSM changes to state *gnt*1, regardless of the other inputs. The arc with the label $\bar{r}_1 r_2$ that leads from state *Idle* to *gnt*2 represents the condition $r_1\, r_2 = 01$, while the value of r_3 is irrelevant. There is no standardized scheme for labeling the arcs in state diagrams. Some designers prefer the style of Figure 6.72, while others prefer a style more similar to Figure 6.73.

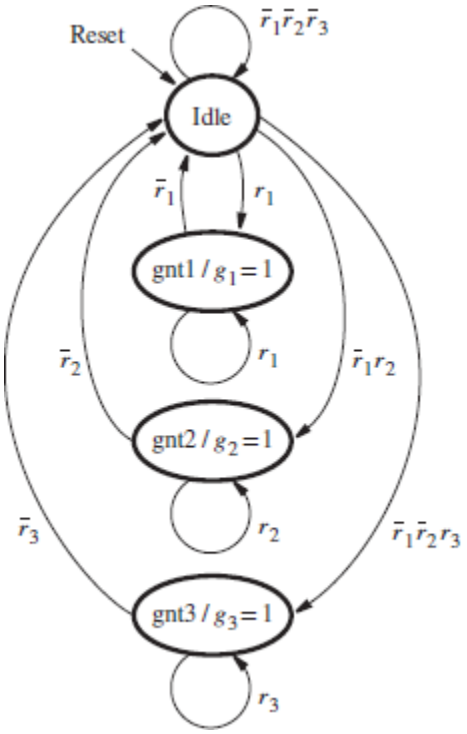


Figure 6.73 Alternative style of state diagram for the arbiter.

Figure 6.74 gives the VHDL code for the machine. The three request and grant signals are specified as three-bit STD_LOGIC_VECTOR signals. The FSM is described using a CASE statement in the style used for Figure 6.29. As shown in the WHEN clause for state *Idle*, it is easy to describe the required priority scheme. The IF statement specifies that if $r_1 = 1$, then the

next state for the machine is *gnt*1. If r_1 is not asserted, then the ELSIF condition is evaluated, which stipulates that if $r_2 = 1$, then the next state will be *gnt*2. Each successive ELSIF clause considers a lower-priority request signal only if all of the higher-priority request signals are not asserted.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY arbiter IS
    PORT ( Clock, Resetn  : IN    STD_LOGIC ;
           r              : IN    STD_LOGIC_VECTOR(1 TO 3) ;
           g              : OUT   STD_LOGIC_VECTOR(1 TO 3) ) ;
END arbiter ;

ARCHITECTURE Behavior OF arbiter IS
    TYPE State_type IS (Idle, gnt1, gnt2, gnt3) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN y <= Idle ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN Idle =>
                    IF r(1) = '1' THEN y <= gnt1 ;
                    ELSIF r(2) = '1' THEN y <= gnt2 ;
                    ELSIF r(3) = '1' THEN y <= gnt3 ;
                    ELSE y <= Idle ;
                    END IF ;
                WHEN gnt1 =>
                    IF r(1) = '1' THEN y <= gnt1 ;
                    ELSE y <= Idle ;
                    END IF ;
                WHEN gnt2 =>
                    IF r(2) = '1' THEN y <= gnt2 ;
                    ELSE y <= Idle ;
                    END IF ;
                WHEN gnt3 =>
                    IF r(3) = '1' THEN y <= gnt3 ;
                    ELSE y <= Idle ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
    g(1) <= '1' WHEN y = gnt1 ELSE '0' ;
    g(2) <= '1' WHEN y = gnt2 ELSE '0' ;
    g(3) <= '1' WHEN y = gnt3 ELSE '0' ;
END Behavior ;
```

Figure 6.74 VHDL code for the arbiter.

The WHEN clause for each grant state is straightforward. For state *gnt*1 it specifies that as long as $r_1 = 1$, the next state remains *gnt*1. When $r_1 = 0$, the next state is *Idle*. The other grant states have the same structure.

The code for the grant signals, g_1, g_2, and g_3 is given at the end. It sets g_1 to 1 when the machine is in state *gnt*1, and otherwise g_1 is set to 0. Similarly, each of the other grant signals is 1 only in the appropriate grant state.

Instead of the three conditional assignment statements used for g_1, g_2, and g_3, it may seem reasonable to substitute the process shown in Figure 6.75, which contains an IF statement. This code is incorrect, but the reason is not obvious. Recall from the discussion concerning Figure 4.43 that when using an IF statement, if there is no ELSE clause or default value for a signal, then that signal retains its value when the IF condition is not met. This is called implied memory. In Figure 6.75 the signal g_1 is set to 1 when the FSM first enters state *gnt*1, and then g_1 will retain the value 1 no matter what state the FSM changes to. Similarly, the code for g_2 and g_3 is also incorrect. If we wish to write the code using an IF statement, then it should be structured as shown in Figure 6.76. Each grant signal is assigned a default value of 0, which avoids the problem of implied memory.

```
.
.
.
PROCESS( y )
BEGIN
    IF y = gnt1 THEN g(1) <= '1' ;
    ELSIF y = gnt2 THEN g(2) <= '1' ;
    ELSIF y = gnt3 THEN g(3) <= '1' ;
    END IF ;
END PROCESS ;
END Behavior ;
```

Figure 6.75 Incorrect VHDL code for the grant signals.

```
.
.
.
PROCESS( y )
BEGIN
        g(1) <= '0' ;
        g(2) <= '0' ;
        g(3) <= '0' ;
        IF y = gnt1 THEN g(1) <= '1' ;
        ELSIF y = gnt2 THEN g(2) <= '1' ;
        ELSIF y = gnt3 THEN g(3) <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.76 Correct VHDL code for the grant signals.

6.9 Analysis of Synchronous Sequential Circuits

In addition to knowing how to design a synchronous sequential circuit, the designer has to be able to analyze the behavior of an existing circuit. The analysis task is much simpler than the synthesis task. In this section we will show how analysis may be performed.

To analyze a circuit, we simply reverse the steps of the synthesis process. The outputs of the flip-flops represent the present-state variables. Their inputs determine the next state that the circuit will enter. From this information we can construct the state-assigned table for the circuit. This table leads to a state table and the corresponding state diagram by giving a name to each state. The type of flip-flops used in the circuit is a factor, as we will see in the examples that follow.

Example 6.8

D-Type Flip-Flops Figure 6.77 gives an FSM that has two D flip-flops. Let y_1 and y_2 be the present-state variables and Y_1 and Y_2 the next-state variables. The next-state and output expressions are

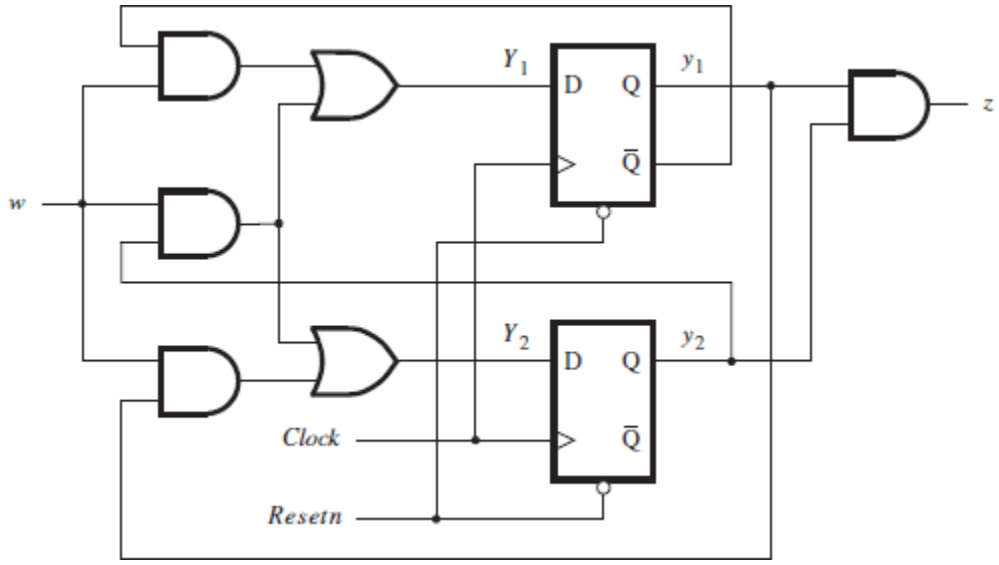


Figure 6.77 Circuit for Example 6.8.

$$Y_1 = w\bar{y}_1 + wy_2$$
$$Y_2 = wy_1 + wy_2$$
$$z = y_1y_2$$

Since there are two flip-flops, the FSM has at most four states. A good starting point in the analysis is to assume an initial state of the flip-flops such as $y_1 = y_2 = 0$. From the expressions for Y_1 and Y_2, we can derive the state-assigned table in Figure 6.78*a*. For example, in the first row of the table $y_1 = y_2 = 0$. Then $w = 0$ causes $Y_1 = Y_2 = 0$, and $w = 1$ causes $Y_1 = 1$ and $Y_2 = 0$. The output for this state is $z = 0$. The other rows are derived in the same manner. Labeling the states as *A*, *B*, *C*, and *D* yields the state table in Figure 6.78*b*. From this table it is apparent that following the reset condition the FSM produces the output $z = 1$ whenever three consecutive 1s occur on the input *w*. Therefore, the FSM acts as a sequence detector for this pattern.

Present state y_2y_1	Next state $w = 0$ Y_2Y_1	Next state $w = 1$ Y_2Y_1	Output z
0 0	0 0	0 1	0
0 1	0 0	1 0	0
1 0	0 0	1 1	0
1 1	0 0	1 1	1

(a) State-assigned table

Present state	Next state $w = 0$	Next state $w = 1$	Output z
A	A	B	0
B	A	C	0
C	A	D	0
D	A	D	1

(b) State table

Figure 6.78 Tables for the circuit in Figure 6.77.

Example 6.9

JK-Type Flip-Flops Now consider the circuit in Figure 6.79, which has two JK flip-flops. The expressions for the inputs to the flip-flops are

$$J_1 = w$$
$$K_1 = \overline{w} + \overline{y}_2$$
$$J_2 = wy_1$$
$$K_2 = \overline{w}$$

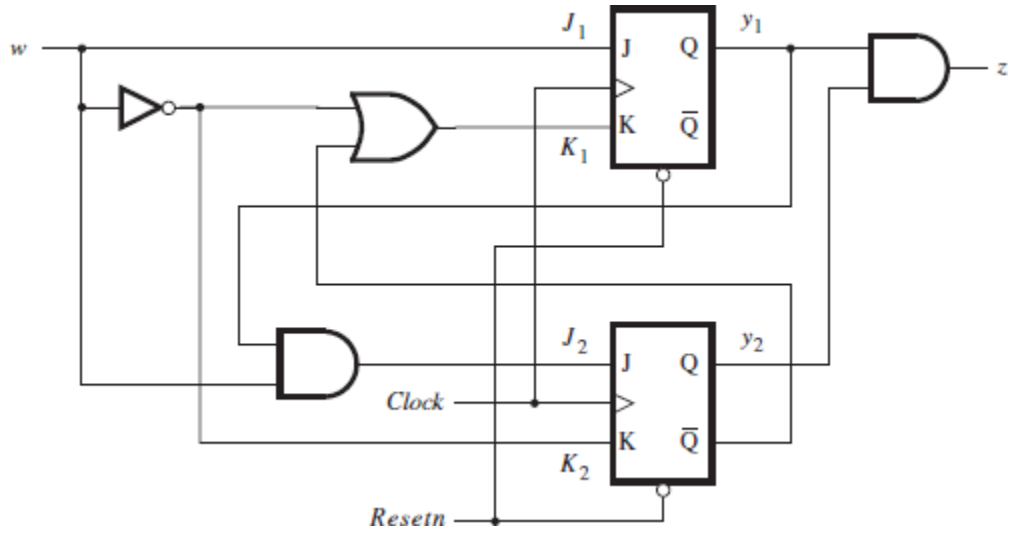


Figure 6.79 Circuit for Example 6.9.

The output is given by $z = y_1 y_2$.

From these expressions we can derive the excitation table in Figure 6.80. Interpreting the entries in this table, we can construct the state-assigned table. For example, consider $y_2 y_1 = 0\ 0$ and $w = 0$. Then, since $J_2 = J_1 = 0$ and $K_2 = K_1 = 1$, both flip-flops will remain in the 0 state; hence $Y_2 = Y_1 = 0$. If $y_2 y_1 = 0\ 0$ and $w = 1$, then $J_2 = K_2 = 0$ and $J_1 = K_1 = 1$, which leaves the y_2 flip-flop unchanged and sets the y_1 flip-flop to 1; hence $Y_2 = 0$ and $Y_1 = 1$. If $y_2 y_1 = 0\ 1$ and $w = 0$, then $J_2 = J_1 = 0$ and $K_2 = K_1 = 1$, which resets the y_1 flip-flop and results in the state $y_2 y_1 = 0\ 0$; hence $Y_2 = Y_1 = 0$. Similarly, if $y_2 y_1 = 0\ 1$ and $w = 1$, then $J_2 = 1$ and $K_2 = 0$ sets y_2 to 1; hence $Y_2 = 1$, while $J_1 = K_1 = 1$ toggles y_1; hence $Y_1 = 0$. This leads to the state $y_2 y_1 = 1\ 0$. Completing this process, we find that the resulting state-assigned table is the same as the one in Figure 6.78*a*. The conclusion is that the circuits in Figures 6.77 and 6.79 implement the same FSM.

Present state y_2y_1	Flip-flop inputs				Output z
	$w = 0$		$w = 1$		
	J_2K_2	J_1K_1	J_2K_2	J_1K_1	
0 0	0 1	0 1	0 0	1 1	0
0 1	0 1	0 1	1 0	1 1	0
1 0	0 1	0 1	0 0	1 0	0
1 1	0 1	0 1	1 0	1 0	1

Figure 6.80 The excitation table for the circuit in Figure 6.79.

Example 6.10

Mixed Flip-Flops There is no reason why one cannot use a mixture of flip-flop types in one circuit. Figure 6.81 shows a circuit with one D and one T flip-flop. The expressions for this circuit are

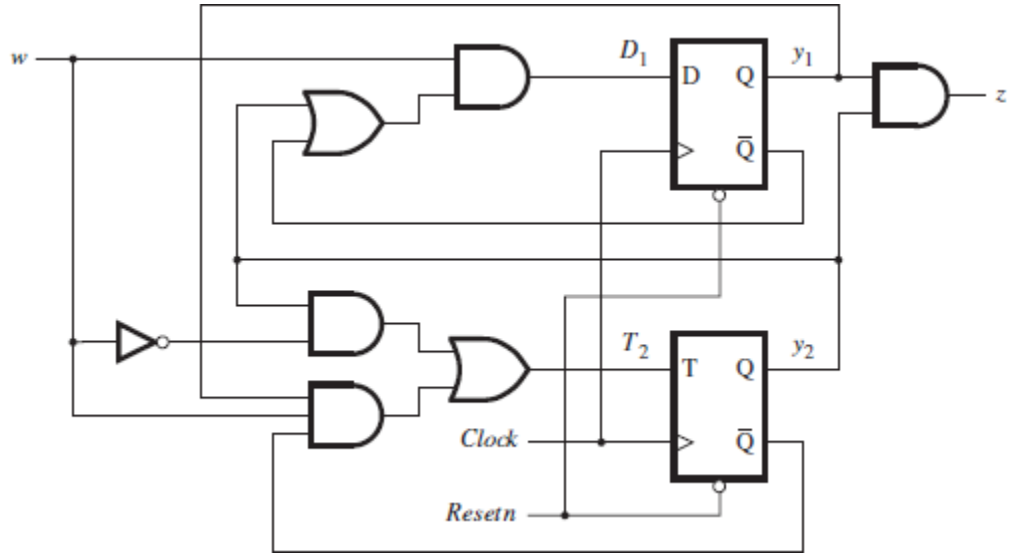


Figure 6.81 Circuit for Example 6.10.

$$D_1 = w(\overline{y}_1 + y_2)$$
$$T_2 = \overline{w}y_2 + wy_1\overline{y}_2$$
$$z = y_1y_2$$

From these expressions we derive the excitation table in Figure 6.82. Since it is a T flip-flop, y_2 changes its state only when $T_2 = 1$. Thus if $y_2\ y_1 = 0\ 0$ and $w = 0$, then because $T_2 = D_1 = 0$ the state of the circuit will not change. An example of where $T_2 = 1$ is when $y_2\ y_1 = 0\ 1$ and $w = 1$, which causes y_2 to change to 1; $D_1 = 0$ makes $y_1 = 0$, hence $Y_2 = 1$ and $Y_1 = 0$. The other cases where $T_2 = 1$ occur when $w = 0$ and $y_2\ y_1 = 10$ or 11. In both of these cases $D_1 = 0$. Hence the T flip-flop changes its state from 1 to 0, while the D flip-flop is cleared, which means that the next state is $Y_2\ Y_1 = 00$.

Completing this analysis we again obtain the state-assigned table in Figure 6.78*a*. Thus this circuit is yet another implementation of the FSM represented by the state table in Figure 6.78*b*.

Present state y_2y_1	Flip-flop inputs		Output z
	$w = 0$	$w = 1$	
	T_2D_1	T_2D_1	
0 0	0 0	0 1	0
0 1	0 0	1 0	0
1 0	1 0	0 1	0
1 1	1 0	0 1	1

Figure 6.82 The excitation table for the circuit in Figure 6.81.

6.10 Algorithmic State Machine (ASM) Charts

The state diagrams and tables used in this chapter are convenient for describing the behavior of FSMs that have only a few inputs and outputs. For larger machines the designers often use a different form of representation, called the *algorithmic state machine (ASM) chart*.

An ASM chart is a type of flowchart that can be used to represent the state transitions and generated outputs for an FSM. The three types of elements used in ASM charts are depicted in Figure 6.83.

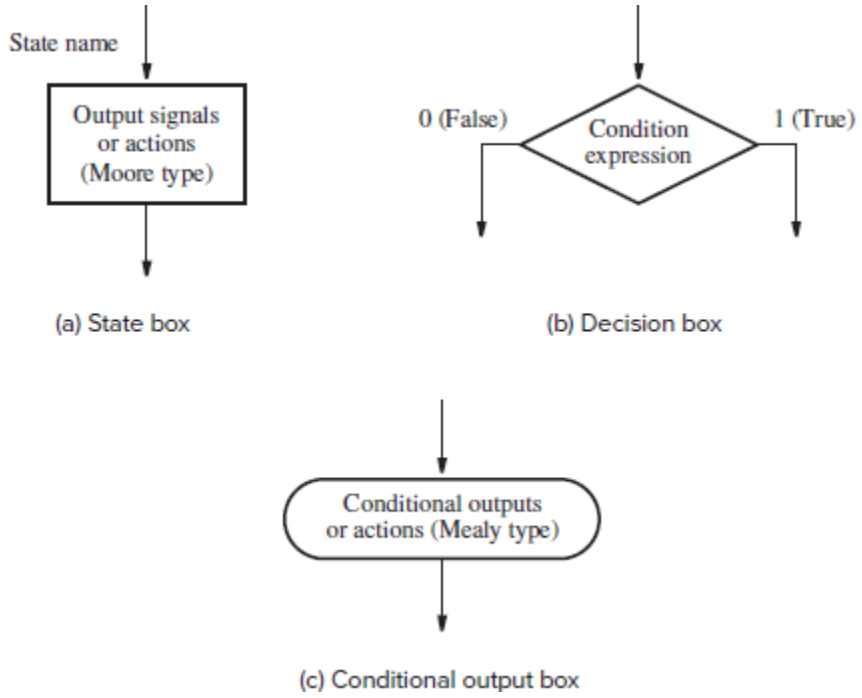


Figure 6.83 Elements used in ASM charts.

- **State box** A rectangle represents a state of the FSM. It is equivalent to a node in the state diagram or a row in the state table. The name of the state is indicated outside the box in the top-left corner. The Moore-type outputs are listed inside the box. These are the outputs that depend only on the values of the state variables that define the state; we will refer to them simply as *Moore outputs*. It is customary to write only the name of the signal that has to be asserted. Thus it is sufficient to write z, rather than $z = 1$, to indicate that the output z must have the value 1. Also, it may be useful to indicate an action that must be taken; for example, $Count \leftarrow Count + 1$ specifies that the contents of a counter have to be incremented by 1. Of course, this is just a simple way of saying that the control signal that causes the counter to be incremented must be asserted. We will use this way of specifying actions in larger systems that are discussed in Chapter 7.
- **Decision box** A diamond indicates that the stated condition expression is to be tested and the exit path is to be chosen accordingly. The condition expression consists of one or more inputs to the FSM. For example, w indicates that the decision is based on the value of the input

w, whereas $w_1 \cdot w_2$ indicates that the true path is taken if $w_1 = w_2 = 1$ and the false path is taken otherwise.

- **Conditional output box** An oval denotes the output signals that are of Mealy type. These outputs depend on the values of the state variables and the inputs of the FSM; we will refer to these outputs simply as *Mealy outputs*. The condition that determines whether such outputs are generated is specified in a decision box.

Figure 6.84 gives the ASM chart that represents the FSM in Figure 6.3. The transitions between state boxes depend on the decisions made by testing the value of the input variable w. In each case if $w = 0$, the exit path from a decision box leads to state A. If $w = 1$, then a transition from A to B or from B to C takes place. If $w = 1$ in state C, then the FSM stays in that state. The chart specifies a Moore output z, which is asserted only in state C, as indicated in the state box. In states A and B, the value of z is 0 (not asserted), which is implied by leaving the corresponding state boxes blank.

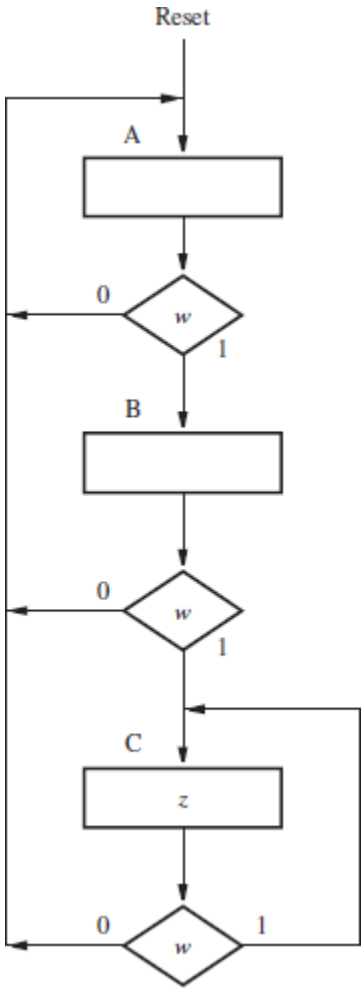


Figure 6.84 ASM chart for the FSM in Figure 6.3.

Figure 6.85 provides an example with Mealy outputs. This chart represents the FSM in Figure 6.23. The output, z, is equal to 1 when the machine is in state B and $w = 1$. This is indicated using the conditional output box. In all other cases the value of z is 0, which is implied by not specifying z as an output of state B for $w = 0$ and state A for w equal to 0 or 1.

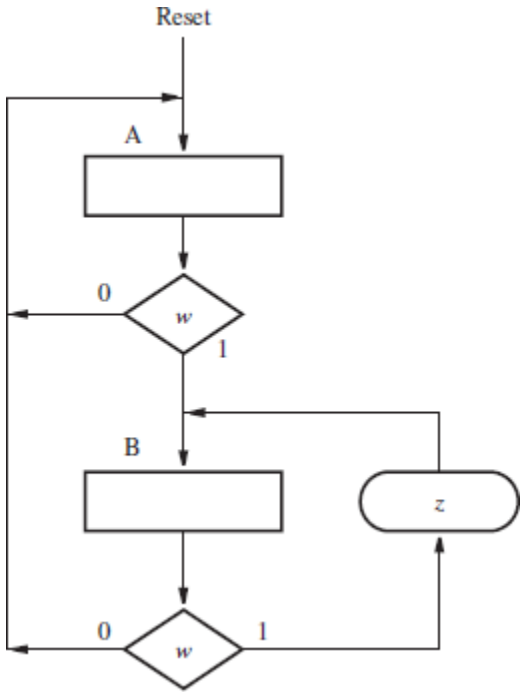


Figure 6.85 ASM chart for the FSM in Figure 6.23.

Figure 6.86 gives the ASM chart for the arbiter FSM in Figure 6.73. The decision box drawn below the state box for *Idle* specifies that if $r_1 = 1$, then the FSM changes to state *gnt1*. In this state the FSM asserts the output signal g_1. The decision box to the right of the state box for *gnt1* specifies that as long as $r_1 = 1$, the machine stays in state *gnt1*, and when $r_1 = 0$, it changes to state *Idle*. The decision box labeled r_2 that is drawn below the state box for *Idle* specifies that if $r_2 = 1$, then the FSM changes to state *gnt2*. This decision box can be reached only after first checking the value of r_1 and following the arrow that corresponds to $r_1 = 0$. Similarly, the decision box labeled r_3 can be reached only if both r_1 and r_2 have the value 0. Hence the ASM chart describes the required priority scheme for the arbiter.

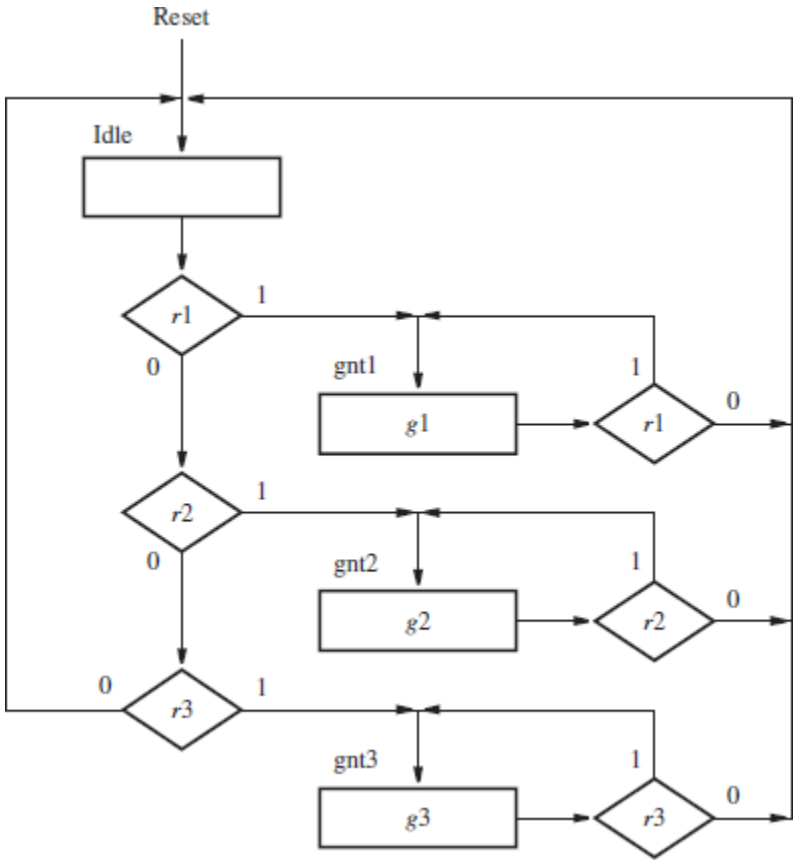


Figure 6.86 ASM chart for the arbiter FSM in Figure 6.73.

ASM charts are similar to traditional flowcharts. Unlike a traditional flowchart, the ASM chart includes timing information because it implicitly specifies that the FSM changes (flows) from one state to another only after each active clock edge. The examples of ASM charts presented here are quite simple. We have used them to introduce the ASM chart terminology by giving examples of state, decision, and conditional-output boxes. Another term sometimes applied to ASM charts is *ASM block*, which refers to a single state box and any decision and conditional-output boxes that the state box may be connected to. The ASM charts can be used to describe complex circuits that include one or more finite state machines and other circuitry such as registers, shift registers, counters, adders, and multipliers. We will use ASM charts as an aid for designing more complex circuits in Chapter 7.

6.11 Formal Model for Sequential Circuits

This chapter has presented the synchronous sequential circuits using a rather informal approach because this is the easiest way to grasp the concepts that are essential in designing such circuits. The same topics can also be presented in a more formal manner, which has been the style adopted in many books that emphasize the switching theory aspects rather than the design using CAD tools. A formal model often gives a concise specification that is difficult to match in a more descriptive presentation. In this section we will describe a formal model that represents a general class of sequential circuits, including those of the synchronous type.

Figure 6.87 represents a general sequential circuit. The circuit has $W = \{w_1, w_2, \ldots, w_n\}$ inputs, $Z = \{z_1, z_2, \ldots, z_m\}$ outputs, $y = \{y_1, y_2, \ldots, y_k\}$ present-state variables, and $Y = \{Y_1, Y_2, \ldots, Y_k\}$ next-state variables. It can have up to 2^k states, $S = \{S_1, S_2, \ldots, S_{2^k}\}$. There are delay elements in the feedback paths for the state-variables which ensure that y will take the values of Y after a time delay Δ. In the case of synchronous sequential circuits, the delay elements are flip-flops, which change their state on the active edge of a clock signal. Thus the delay Δ is determined by the clock period. The clock period must be long enough to allow for the propagation delay in the combinational circuit, in addition to the setup and hold parameters of the flip-flops.

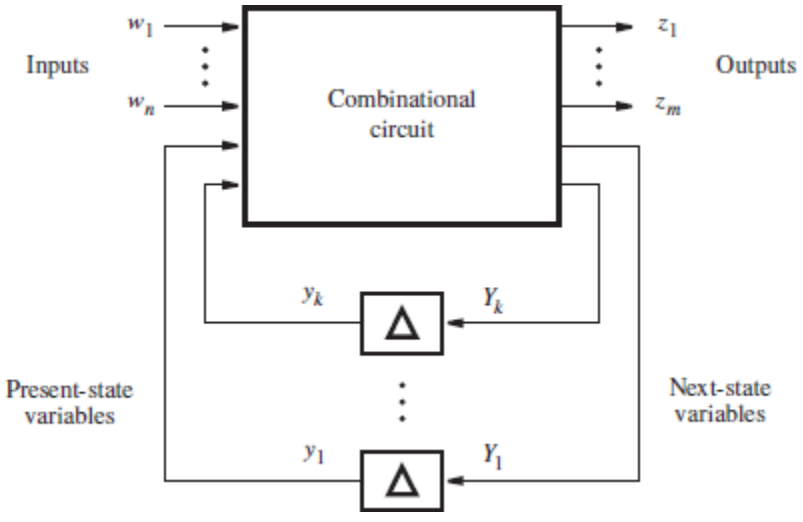


Figure 6.87 The general model for a sequential circuit.

Using the model in Figure 6.87, a synchronous sequential circuit, M, can be defined formally as a quintuple

$$M = (W, Z, S, \varphi, \lambda)$$

where

- W, Z, and S are finite, nonempty sets of inputs, outputs, and states, respectively.
- ϕ is the state transition function, such that $S(t + 1) = \phi\ [W(t), S(t)]$.
- λ is the output function, such that $Z(t) = \lambda\ [S(t)]$ for the Moore model and $Z(t) = \lambda\ [W(t), S(t)]$ for the Mealy model.

This definition assumes that the time between t and t+1 is one clock cycle.

We will see in Chapter 9 that the delay Δ need not be controlled by a clock. In asynchronous sequential circuits the delays are due solely to the propagation delays through various gates.

6.12 Concluding Remarks

The existence of closed loops and delays in a sequential circuit leads to a behavior that is characterized by the set of states that the circuit can reach. The present values of the inputs are not the sole determining factor in this behavior, because a given valuation of inputs may cause the circuit to behave differently in different states.

The propagation delays through a sequential circuit must be taken into account. The design techniques presented in this chapter are based on the assumption that all changes in the circuit are triggered by the active edge of a clock signal. Such circuits work correctly only if all internal signals are stable when the clock signal arrives. Thus the clock period must be longer than the longest propagation delay in the circuit.

Synchronous sequential circuits are used extensively in practical designs. They are supported by the commonly used CAD tools.

In Chapter 9 we will present a different class of sequential circuits, which do not use flip-flops to represent the states of the circuit and do not use clock pulses to trigger changes in the states.

6.13 Examples of Solved Problems

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

Example 6.11

Problem: Design an FSM that has an input w and an output z. The machine is a sequence detector that produces $z = 1$ when the previous two values of w were 00 or 11; otherwise $z = 0$.

Solution: Section 6.1 presents the design of an FSM that detects the occurrence of consecutive 1s. Using the same approach, the desired FSM can be specified using the state diagram in Figure 6.88. State C denotes the occurrence of two or more 0s, and state E denotes two or more 1s. The corresponding state table is shown in Figure 6.89.

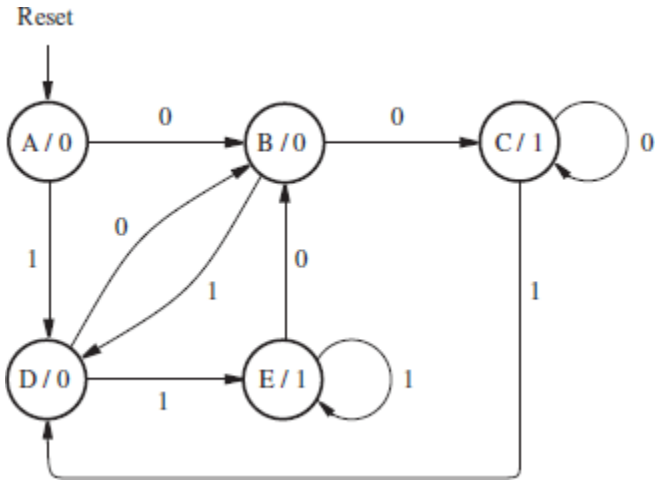


Figure 6.88 State diagram for Example 6.11.

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	B	D	0
B	C	D	0
C	C	D	1
D	B	E	0
E	B	E	1

Figure 6.89 State table for the FSM in Figure 6.88.

A straightforward state assignment leads to the state-assigned table in Figure 6.90. The codes $y_3\,y_2\,y_1$ = 101, 110, 111 can be treated as don't-care conditions. Then the next-state expressions are

	Present state $y_3y_2y_1$	Next state $w = 0$ $Y_3Y_2Y_1$	Next state $w = 1$ $Y_3Y_2Y_1$	Output z
A	000	001	011	0
B	001	010	011	0
C	010	010	011	1
D	011	001	100	0
E	100	001	100	1

Figure 6.90 State-assigned table for the FSM in Figure 6.89.

$$Y_1 = w\bar{y}_1\bar{y}_3 + w\bar{y}_2\bar{y}_3 + \bar{w}y_1y_2 + \overline{wy}_1\bar{y}_2$$
$$Y_2 = y_1\bar{y}_2 + \bar{y}_1y_2 + w\bar{y}_2\bar{y}_3$$
$$Y_3 = wy_3 + wy_1y_2$$

The output expression is

$$z = y_3 + \bar{y}_1y_2$$

These expressions seem to be unnecessarily complex, suggesting that we may attempt to find a better state assignment. Observe that state *A* is reached only when the machine is reset by means of the *Reset* input. So, it may be advantageous to assign the four codes in which y_3 = 1 to the states *B*, *C*, *D*, and *E*. The result is the state-assigned table in Figure 6.91. From it, the next-state and output expressions are

	Present state $y_3y_2y_1$	Next state $w = 0$ $Y_3Y_2Y_1$	Next state $w = 1$ $Y_3Y_2Y_1$	Output z
A	000	100	110	0
B	100	101	110	0
C	101	101	110	1
D	110	100	111	0
E	111	100	111	1

Figure 6.91 An improved state assignment for the FSM in Figure 6.89.

$$Y_1 = wy_2 + \overline{w}y_3\overline{y}_2$$
$$Y_2 = w$$
$$Y_3 = 1$$
$$z = y_1$$

Since $Y_3 = 1$, we do not need the flip-flop y_3, and the expression for Y_1 simplifies to $\overline{w \oplus y_2}$. This is a much better solution.

Example 6.12

Problem: Implement the sequence detector of Example 6.11 by using two FSMs. One FSM detects the occurrence of consecutive 1s, while the other detects consecutive 0s.

Solution: A good realization of the FSM that detects consecutive 1s is given in Figure 6.17. The next-state and output expressions are

$$Y_1 = w$$
$$Y_2 = wy_1$$
$$z_{ones} = y_2$$

A similar FSM that detects consecutive 0s is defined in Figure 6.92. Its expressions are

Present state	Next state		Output
	$w = 0$	$w = 1$	z_{zeros}
D	E	D	0
E	F	D	0
F	F	D	1

(a) State table

	Present state	Next state		Output
		$w = 0$	$w = 1$	
	y_4y_3	Y_4Y_3	Y_4Y_3	z_{zeros}
D	00	01	00	0
E	01	11	00	0
F	11	11	00	1
	10	*dd*	*dd*	*d*

(b) State-assigned table

Figure 6.92 FSM that detects a sequence of two zeros.

$$Y_3 = \overline{w}$$
$$Y_4 = \overline{w}y_3$$
$$z_{zeros} = y_4$$

The output of the combined circuit is

$$z = z_{ones} + z_{zeros}$$

Example 6.13

Problem: Derive a Mealy-type FSM that can act as a sequence detector described in Example 6.11.

Solution: A state diagram for the desired FSM is depicted in Figure 6.93. The corresponding state table is presented in Figure 6.94. Two flip-flops are needed to implement this FSM. A state-assigned table is given in Figure 6.95, which leads to the next-state and output expressions

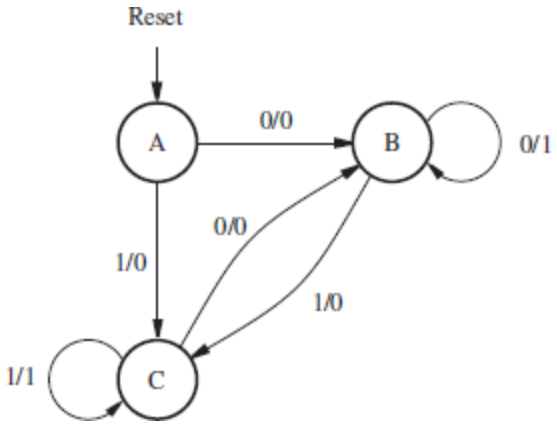


Figure 6.93 State diagram for Example 6.13.

$$Y_1 = 1$$
$$Y_2 = w$$
$$z = \overline{w}y_1\overline{y}_2 + wy_2$$

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	B	C	1	0
C	B	C	0	1

Figure 6.94 State table for the FSM in Figure 6.93.

	Present state	Next state		Output	
		$w = 0$	$w = 1$	$w = 0$	$w = 1$
	y_2y_1	Y_2Y_1	Y_2Y_1	z	z
A	00	01	11	0	0
B	01	01	11	1	0
C	11	01	11	0	1

Figure 6.95 State-assigned table for the FSM in Figure 6.94.

Example 6.14

Problem: Implement the state-assigned table in Figure 6.91 using JK-type flip-flops.

Solution: Figure 6.96 shows the excitation table. It results in the following next-state and output expressions

$$J_1 = wy_2 + \overline{w}y_3\overline{y}_2$$
$$K_1 = \overline{w}y_2 + wy_1\overline{y}_2$$
$$J_2 = w$$
$$K_2 = \overline{w}$$
$$J_3 = 1$$
$$K_3 = 0$$
$$z = y_1$$

	Present state $y_3y_2y_1$	Flip-flop inputs								Output z
		$w = 0$				$w = 1$				
		$Y_3Y_2Y_1$	J_3K_3	J_2K_2	J_1K_1	$Y_3Y_2Y_1$	J_3K_3	J_2K_2	J_1K_1	
A	000	100	1*d*	0*d*	0*d*	110	1*d*	1*d*	0*d*	0
B	100	101	*d*0	0*d*	1*d*	110	*d*0	1*d*	0*d*	0
C	101	101	*d*0	0*d*	*d*0	110	*d*0	1*d*	*d*1	1
D	110	100	*d*0	*d*1	0*d*	111	*d*0	*d*0	1*d*	0
E	111	100	*d*0	*d*1	*d*1	111	*d*0	*d*0	*d*0	1

Figure 6.96 Excitation table for the FSM in Figure 6.91 with JK flip-flops.

Example 6.15

Problem: Write VHDL code to implement the FSM in Figure 6.88.

Solution: Using the style of code given in Figure 6.29, the required FSM can be specified as shown in Figure 6.97.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY sequence IS
    PORT ( Clock, Resetn, w     : IN     STD_LOGIC ;
           z                    : OUT  STD_LOGIC ) ;
END sequence ;

ARCHITECTURE Behavior OF sequence IS
    TYPE State_type IS (A, B, C, D, E) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= D ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN y <= C ;
                    ELSE y <= D ;
                    END IF ;
                WHEN C =>
                    IF w = '0' THEN y <= C ;
                    ELSE y <= D ;
                    END IF ;
                WHEN D =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= E ;
                    END IF ;
                WHEN E =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= E ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
    z <= '1' WHEN (y = C OR y = E) ELSE '0' ;
END Behavior ;
```

Figure 6.97 VHDL code for the FSM in Figure 6.88.

Example 6.16

Problem: Write VHDL code to implement the FSM in Figure 6.93.

Solution: Using the style of code given in Figure 6.36, the Mealy-type FSM can be specified as shown in Figure 6.98.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY seqmealy IS
    PORT ( Clock, Resetn, w     : IN    STD_LOGIC ;
           z                    : OUT   STD_LOGIC ) ;
END seqmealy ;

ARCHITECTURE Behavior OF seqmealy IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= C ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= C ;
                    END IF ;
                WHEN C =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= C ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    PROCESS ( y, w )
    BEGIN
        CASE y IS
                WHEN A =>
                    z <= '0' ;
                WHEN B =>
                    z <= NOT w ;
                WHEN C =>
                    z <= w ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 6.98 VHDL code for the FSM in Figure 6.93.

Example 6.17

Problem: In computer systems it is often desirable to transmit data serially, namely, one bit at a time, to save on the cost of interconnecting cables. This means that parallel data at one end must be transmitted serially, and at the other end the received serial data has to be turned back into parallel form. Suppose that we wish to transmit ASCII characters in this manner. As explained in Chapter 1, the standard ASCII code uses seven bits to define each character. Usually, a character occupies one byte, in which case the eighth bit can either be set to 0 or it can be used to indicate the parity of the other bits to ensure a more reliable transmission.

Parallel-to-serial conversion can be done by means of a shift register. Assume that a circuit accepts parallel data, $B = b_7, b_6, \ldots, b_0$, representing ASCII characters. Assume also that bit b_7 is set to 0. The circuit is supposed to generate a parity bit, p, and send it instead of b_7 as a part of the serial transfer. Figure 6.99 gives a possible circuit. An FSM is used to generate the parity bit, which is included in the output stream by using a multiplexer. A three-bit counter is used to determine when the p bit is transmitted, which happens when the count reaches 7. Design the desired FSM.

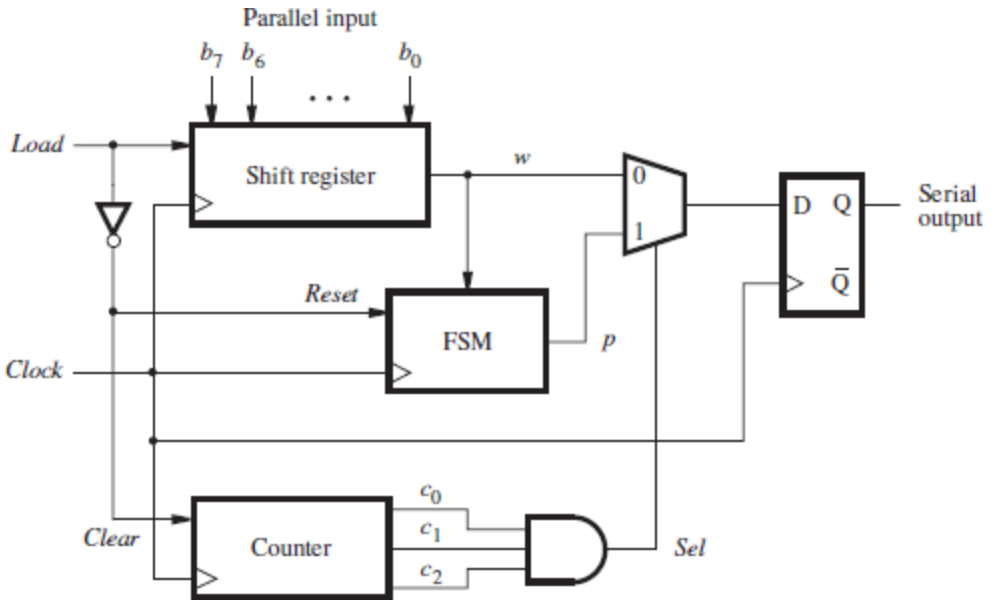


Figure 6.99 Parallel-to-serial converter.

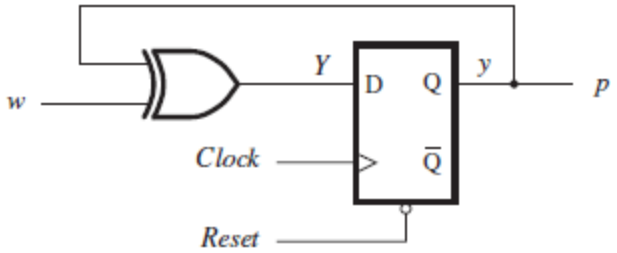
Solution: As the bits are shifted out of the shift register, the FSM examines the bits and keeps track of whether there has been an even or odd number of 1s. It sets *p* to 1 if there is odd parity. Hence, the FSM must have two states. Figure 6.100 presents the state table, the state-assigned table, and the resulting circuit. The next state expression is

Present state	Next state		Output
	$w = 0$	$w = 1$	p
S_{even}	S_{even}	S_{odd}	0
S_{odd}	S_{odd}	S_{even}	1

(a) State table

Present state	Next state		Output
	$w = 0$	$w = 1$	p
y	Y	Y	
0	0	1	0
1	1	0	1

(b) State-assigned table



(c) Circuit

Figure 6.100 FSM for parity generation.

$$Y = \overline{w}y + w\overline{y}$$

The output *p* is just equal to *y*.

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

***6.1** An FSM is defined by the state-assigned table in Figure P6.1. Derive a circuit that realizes this FSM using D flip-flops.

Present state y_2y_1	Next state $w = 0$ Y_2Y_1	Next state $w = 1$ Y_2Y_1	Output z
0 0	1 0	1 1	0
0 1	0 1	0 0	0
1 0	1 1	0 0	0
1 1	1 0	0 1	1

Figure P6.1 State-assigned table for Problems 6.1 and 6.2.

***6.2** Derive a circuit that realizes the FSM defined by the state-assigned table in Figure P6.1 using JK flip-flops.

6.3 Derive the state diagram for an FSM that has an input w and an output z. The machine has to generate $z = 1$ when the previous four values of w were 1001 or 1111; otherwise, $z = 0$. Overlapping input patterns are allowed. An example of the desired behavior is

w : 0101111001100111111
z : 0000001001000100 11

6.4 Write VHDL code for the FSM described in Problem 6.3.

***6.5** Derive a minimal state table for a single-input and single-output Moore-type FSM that produces an output of 1 if in the input sequence it detects either 110 or 101 patterns. Overlapping sequences should be detected.

***6.6** Repeat Problem 6.5 for a Mealy-type FSM.

6.7 Derive the circuits that implement the state tables in Figures 6.51 and 6.52. What is the effect of state minimization on the cost of implementation?

6.8 Derive the circuits that implement the state tables in Figures 6.55 and 6.56. Compare the costs of these circuits.

6.9 A sequential circuit has two inputs, w_1 and w_2, and an output, z. Its function is to compare the input sequences on the two inputs. If $w_1 = w_2$ during any four consecutive clock cycles, the circuit produces $z = 1$; otherwise, $z = 0$. For example

w_1 : 0110111000110
w_2 : 1110101000111
z : 0000100001110

Derive a suitable circuit.

6.10 Write VHDL code for the FSM described in Problem 6.9.

6.11 A given FSM has an input, w, and an output, z. During four consecutive clock pulses, a sequence of four values of the w signal is applied. Derive a state table for the FSM that produces $z = 1$ when it detects that either the sequence w : 0010 or w : 1110 has been applied; otherwise, $z = 0$. After the fourth clock pulse, the machine has to be again in the reset state, ready for the next sequence. Minimize the number of states needed.

***6.12** Derive a minimal state table for an FSM that acts as a three-bit parity generator. For every three bits that are observed on the input w during three consecutive clock cycles, the FSM generates the parity bit $p = 1$ if and only if the number of 1s in the three-bit sequence is odd.

6.13 Write VHDL code for the FSM described in Problem 6.12.

6.14 Draw timing diagrams for the circuits in Figures 6.43 and 6.47, assuming the same changes in a and b signals for both circuits. Account for propagation delays.

***6.15** Show a state table for the state-assigned table in Figure P6.1, using A, B, C, D for the four rows in the table. Give a new state-assigned table using a one-hot encoding. For A use the code $y_4\, y_3\, y_2\, y_1 = 0001$. For states B, C, D use the codes 0010, 0100, and 1000, respectively. Synthesize a circuit using D flip-flops.

6.16 Show how the circuit derived in Problem 6.15 can be modified such that the code $y_4\, y_3\, y_2\, y_1 = 0000$ is used for the reset state, A, and the other codes for state B, C, D are changed as needed. (*Hint:* You do not have to resynthesize the circuit!)

***6.17** In Figure 6.59 assume that the unspecified outputs in states *B* and *G* are 0 and 1, respectively. Derive the minimized state table for this FSM.

6.18 In Figure 6.59 assume that the unspecified outputs in states *B* and *G* are 1 and 0, respectively. Derive the minimized state table for this FSM.

6.19 Derive circuits that implement the FSMs defined in Figures 6.57 and 6.58. Can you draw any conclusions about the complexity of circuits that implement Moore and Mealy types of machines?

6.20 Design a counter that counts pulses on line *w* and displays the count in the sequence 0, 2, 1, 3, 0, 2, …. Use D flip-flops in your circuit.

***6.21** Repeat Problem 6.20 using JK flip-flops.

***6.22** Repeat Problem 6.20 using T flip-flops.

6.23 Design a modulo-6 counter, which counts in the sequence 0, 1, 2, 3, 4, 5, 0, 1, …. The counter counts the clock pulses if its enable input, *w*, is equal to 1. Use D flip-flops in your circuit.

6.24 Repeat Problem 6.23 using JK flip-flops.

6.25 Repeat Problem 6.23 using T flip-flops.

6.26 Design a three-bit counterlike circuit controlled by the input *w*. If $w = 1$, then the counter adds 2 to its contents, wrapping around if the count reaches 8 or 9. Thus if the present state is 8 or 9, then the next state becomes 0 or 1, respectively. If $w = 0$, then the counter subtracts 1 from its contents, acting as a normal down-counter. Use D flip-flops in your circuit.

6.27 Repeat Problem 6.26 using JK flip-flops.

6.28 Repeat Problem 6.26 using T flip-flops.

***6.29** Derive the state table for the circuit in Figure P6.2. What sequence of input values on wire *w* is detected by this circuit?

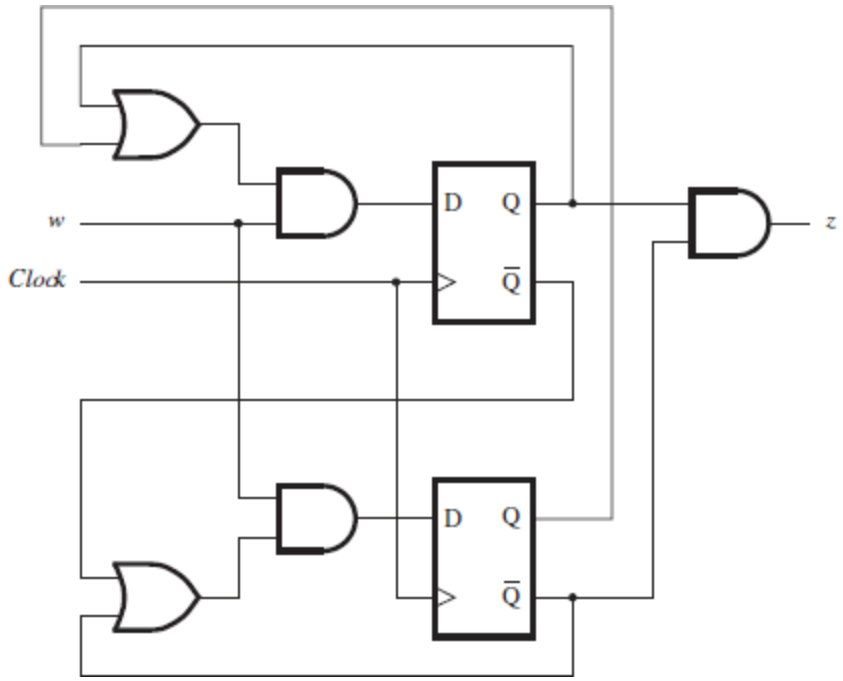


Figure P6.2 Circuit for Problem 6.29.

6.30 Write VHDL code for the FSM shown in Figure 6.57, using the style of code in Figure 6.29.

6.31 Repeat Problem 6.30, using the style of code in Figure 6.33.

6.32 Write VHDL code for the FSM shown in Figure 6.58, using the style of code in Figure 6.29.

6.33 Repeat Problem 6.32, using the style of code in Figure 6.33.

6.34 Write VHDL code for the FSM shown in Figure P6.1.

6.35 Represent the FSM in Figure 6.57 in form of an ASM chart.

6.36 Represent the FSM in Figure 6.58 in form of an ASM chart.

6.37 The arbiter FSM defined in Section 6.8 (Figure 6.72) may cause device 3 to never get serviced if devices 1 and 2 continuously keep raising requests, so that in the Idle state it always happens that either device 1 or device 2 has an outstanding request. Modify the proposed FSM to ensure that device 3 will get serviced, such that if it raises a request, the devices 1 and 2 will be serviced only once before the device 3 is granted its request.

6.38 Write VHDL code for the FSM designed in Problem 6.37.

6.39 Write VHDL code to specify the circuit in Figure 6.99.

6.40 Section 6.5 presents a design for the serial adder. Derive a similar circuit that functions as a serial subtractor which produces the

difference of operands *A* and *B*. (*Hint:* Use the rule for finding 2's complements in Section 3.3.1 to generate the 2's complement of *B*.)

6.41 Write VHDL code that defines the serial subtractor designed in Problem 6.40.

6.42 In Section 6.2 we stated that trying all possible state assignments in order to find the best one is impractical. Determine the number of possible state assignments for an FSM that has *n* states and for which $k = log_2 n$ state variables are used.

chapter

7

DIGITAL SYSTEM DESIGN

CHAPTER OBJECTIVES

In this chapter you will learn about aspects of digital system design, including:

- Bus structure
- A simple processor
- Usage of ASM charts
- Clock synchronization and timing issues
- Flip-flop timing at the chip level

In the previous chapters we showed how to design many types of simple circuits, such as multiplexers, decoders, flip-flops, registers, and counters, which can be used as building blocks. In this chapter we provide examples of more complex circuits that can be constructed using the building blocks as subcircuits. Such larger circuits form a *digital system*. For practical reasons our examples of digital systems will not be large, but the design techniques presented are applicable to systems of any size.

A digital system consists of two main parts, called the datapath circuit and the control circuit. The *datapath circuit* is used to store and manipulate data and to transfer data from one part of the system to another. Datapath circuits comprise building blocks such as registers, shift registers, counters, multiplexers, decoders,

adders, and so on. The *control circuit* controls the operation of the datapath circuit. In Chapter 6 we referred to the control circuits as finite state machines. We will give several examples of digital systems and show how to design their datapath and control circuits.

7.1 Bus Structure

Digital systems often contain a set of registers used to store data. As already discussed in Chapter 6, such registers can be interconnected via an interconnection network as illustrated in Figure 6.10. There are many ways of implementing an interconnection network. In this section we will present one of the commonly used schemes.

Suppose that a digital system has a number of *n*-bit registers, and that it must be possible to transfer data from any register to any other register. A simple way of providing the desired interconnectivity may be to connect each register to a common set of *n* wires, which are used to transfer data into and out of the registers. This common set of wires is usually called a *bus*. If common paths are used to transfer data from multiple sources to multiple destinations, it is necessary to ensure that only one register acts as a source at any given time and other registers do not interfere. We will present two plausible arrangements for implementing the bus structure.

7.1.1 Using Tri-State Drivers to Implement a Bus

Outputs of two ordinary logic gates cannot be connected together, because a short circuit would result if one gate forces the output value 1 while the other gate forces the value 0. Therefore, some special gates are needed if the outputs of two registers are to be connected to a common set of wires. A commonly used circuit element for this purpose is shown in Figure 7.1*a*. It has a data input *w*, an output *f*, and an enable input *e*. Its operation is illustrated by the equivalent circuit in part (*b*). The triangular symbol in the figure represents a noninverting *driver*, which is a circuit that performs no logic operation and its output simply replicates the input signal. Its purpose is to provide additional electrical driving capability. In conjunction with the output switch, it behaves as indicated in Figure 7.1*c*. When $e = 1$, the output reflects the logic value on the data input. But, when $e = 0$, the output is electrically disconnected from the data input, which is referred to as a *high impedance* state and it is usually denoted by the letter Z (or z). Because this circuit element exhibits three distinct states, 0, 1, and Z, it is called a *tri-state*

driver (or buffer). Appendix B explains how it can be implemented with transistors.

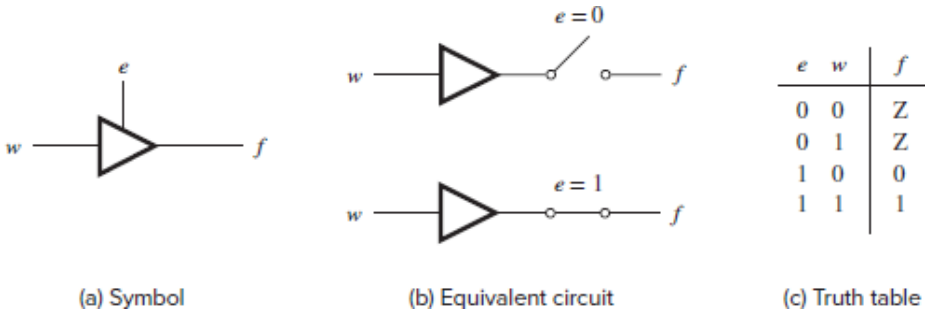


Figure 7.1 Tri-state driver.

Consider a system that contains k n-bit registers, $R1$ to Rk. Figure 7.2 shows how these registers can be connected using tri-state drivers to implement the bus structure. The data outputs of each register are connected to tri-state drivers. When selected by their enable signals, the drivers place the contents of the corresponding register onto the bus wires. We showed in Figure 5.51 how an enable input can be added to a register. If the enable input is set to 1, then the contents of the register will be changed on the next active edge of the clock. In Figure 7.2 the enable input on each register is labeled L, which stands for *load*. The signal that controls the load input for a register Rj is denoted as Rj_{in}, while the signal that controls the associated tri-state driver is called Rj_{out}. These signals are generated by a control circuit.

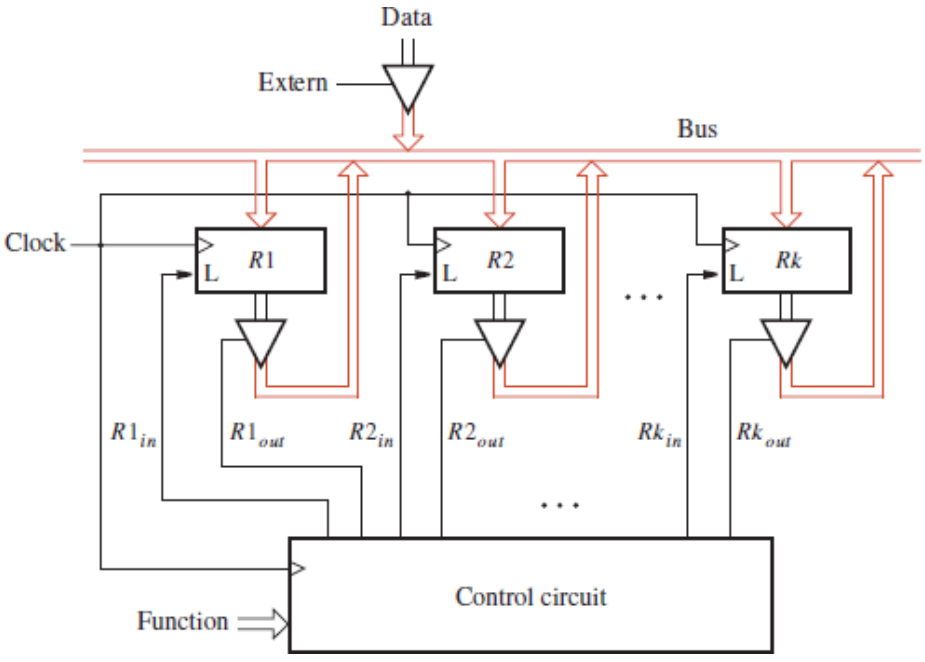


Figure 7.2 A digital system with *k* registers.

In addition to registers, in a real system other types of circuit blocks would be connected to the bus. The figure shows how *n* bits of data from an external source can be placed on the bus, using the signal *Extern*. This signal is also generated by the control circuit.

It is essential to ensure that only one circuit block attempts to place data onto the bus wires at any given time. The control circuit must ensure that only one of the tri-state driver enable signals, $R1_{out}$, …, Rk_{out}, is asserted at once. The control circuit also produces the signals $R1_{in}$, …, Rk_{in}, which determine when data is loaded into each register. In general, the control circuit could perform a number of functions, such as transferring the data stored in one register into another register and controlling the processing of data in various functional units of the system. Figure 7.2 shows an input signal named *Function* that instructs the control circuit to perform a particular task. The control circuit is synchronized by a clock input, which is the same clock signal that controls the *k* registers.

Figure 7.3 provides a more detailed view of how the registers from Figure 7.2 can be connected to a bus. To keep the picture simple, 2 two-bit registers are shown, but the same scheme can be used for larger registers. For register *R*1, two tri-state drivers enabled by $R1_{out}$ are used to connect each flip-flop output to a wire in the bus. The *D* input on each flip-flop is connected to a 2-to-1 multiplexer, whose select input is controlled by $R1_{in}$. If $R1_{in} = 0$, the flip-flops are loaded from their Q outputs; hence the stored data does not change. But if $R1_{in} = 1$, data is loaded into the flip-flops from the bus.

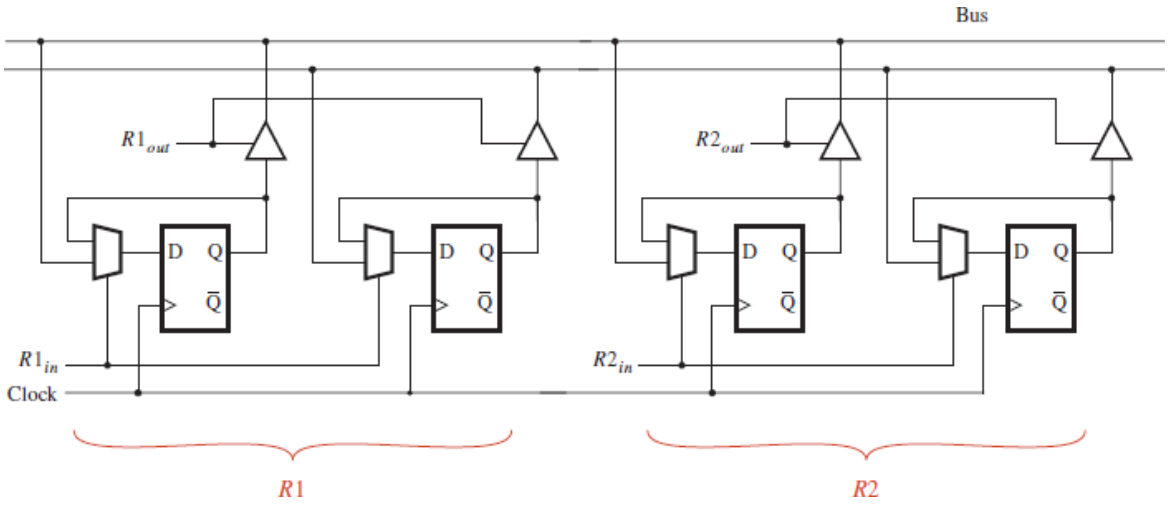


Figure 7.3 Details for connecting registers to a bus.

The system in Figure 7.2 can be used in many different ways, depending on the design of the control circuit and on how many registers and other circuit blocks are connected to the bus. As a simple example, consider a system that has three registers, *R*1, *R*2, and *R*3. We will specify a control circuit that performs a single function—it swaps the contents of registers *R*1 and *R*2, using *R*3 for temporary storage.

The required swapping is done in three steps, each needing one clock cycle. In the first step the contents of *R*2 are transferred into *R*3. Then the contents of *R*1 are transferred into *R*2. Finally, the contents of *R*3, which are the original contents of *R*2, are transferred into *R*1. We have already designed the control circuit for this task in Example 6.1, in the form of a finite state machine.

Note that we say that the contents of one register, R_i, are "transferred" into another register, R_j. This jargon is commonly used to indicate that the new contents of R_j will be a copy of the contents of R_i. The contents of R_i are not changed as a result of the transfer. Therefore, it would be more precise to say that the contents of R_i are "copied" into R_j.

7.1.2 USING MULTIPLEXERS TO IMPLEMENT A BUS

In Figure 7.2 we used tri-state drivers to control access to the bus. An alternative approach is to use multiplexers, as depicted in Figure 7.4. The outputs of each register are connected to a multiplexer. This multiplexer's output is connected to the inputs of the registers, thus realizing the bus. The multiplexer select inputs determine which register's contents appear on the bus. Although the figure shows just one multiplexer symbol, we actually need one multiplexer for each bit in the registers. For example, assume that there are 3 eight-bit registers, *R*1 to *R*3, plus the externally-supplied eight-bit *Data*. To interconnect them, we need eight 4-to-1 multiplexers.

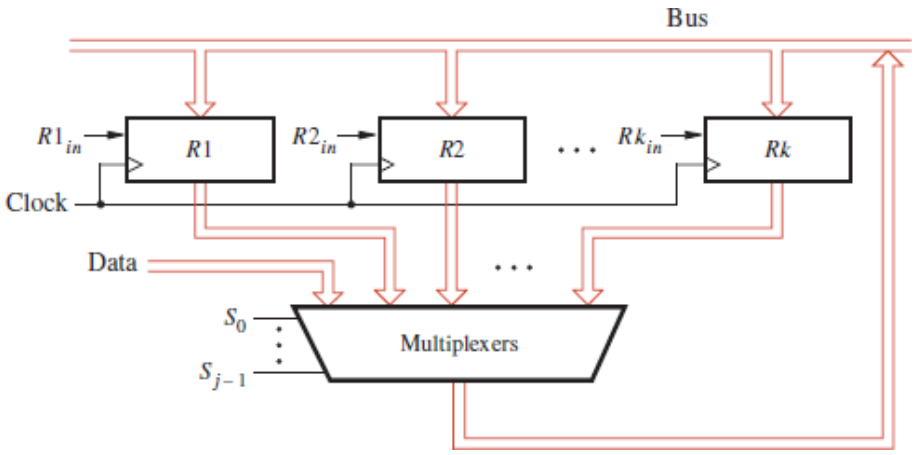


Figure 7.4 Using multiplexers to implement a bus.

The control circuit can be designed as a finite state machine as mentioned in the previous subsection. However, instead of using the control signals Rj_{out} to place the contents of register Rj onto the bus, we have to generate the select inputs for the multiplexers.

The tri-state driver and multiplexer approaches for implementing a bus are both equally valid. However, some types of chips do not contain a sufficient number of tri-state drivers to realize even moderately large buses. In such chips the multiplexer-based approach is the only practical alternative. In practice, circuits are designed with CAD tools. If the designer describes the circuit using tri-state drivers, but there are not enough such drivers in the target device, then the CAD tools may automatically produce an equivalent circuit that uses multiplexers.

7.1.3 VHDL Code for Specification of Bus Structures

This section presents VHDL code for our circuit example that swaps the contents of two registers. We first give the code for the style of circuit in Figure 7.2 that uses tri-state drivers to implement the bus and then give the code for the style of circuit in Figure 7.4 that uses multiplexers. Figure 7.5 gives the code for an n-bit register of the type in Figure 7.3. The number of bits in the register is set by the parameter N, which has the default value of 8. The register is specified such that if the input $L = 1$, then the flip-flops are loaded from the n-bit input R. Otherwise, the flip-flops retain their presently stored values.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT ( R          : IN    STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           L, Clock   : IN    STD_LOGIC ;
           Q          : OUT   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF L = '1' THEN
            Q <= R ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 7.5 Code for an n-bit register of the type in Figure 7.3.

Figure 7.6 gives the code for a subcircuit that represents *n* tri-state drivers, each enabled by the input *E*. The number of drivers is set by the generic parameter *N*. The inputs to the drivers are the *n*-bit signal *X*, and the outputs are the *n*-bit signal *F*. The architecture uses the syntax (OTHERS => 'Z') to specify that the output of each driver is set to the value Z if *E*=0; otherwise, the output is set to *F* = *X*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY trin IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT ( X  : IN   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           E  : IN   STD_LOGIC ;
           F  : OUT  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END trin ;

ARCHITECTURE Behavior OF trin IS
BEGIN
    F <= (OTHERS => 'Z') WHEN E = '0' ELSE X ;
END Behavior ;
```

Figure 7.6 Code for an *n*-bit tri-state driver.

The code in Figure 7.7 represents a digital system like the one in Figure 7.2, with 3 eight-bit registers, *R*1, *R*2, and *R*3. Part (*a*) of the figure shows the control circuit, and part (*b*) provides the datapath. The circuit in Figure 7.2 includes tri-state drivers that are used to place *n* bits of externally supplied *Data* on the bus. In the datapath circuit specified in Figure 7.7*b*, these drivers are instantiated in the component labeled *tri_ext*. Each of the eight drivers is enabled by the signal *Extern*, and the data inputs on the drivers are attached to the eight-bit signal *Data*. When *Extern*=1, the value of *Data* is placed on the bus, which is represented by the signal *BusWires*. The *BusWires* signal is declared with the type INOUT, because it represents the circuit's output as well as the internal bus wiring.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.components.all ;

ENTITY swap IS
    PORT ( Data       : IN    STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Resetn, w  : IN    STD_LOGIC ;
           Clock      : IN    STD_LOGIC ;
           RinExt     : IN    STD_LOGIC_VECTOR(1 TO 3) ;
           BusWires   : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Done       : OUT   STD_LOGIC ) ;
END swap ;

ARCHITECTURE Behavior OF swap IS
    TYPE State_type IS (A, B, C, D) ;
    SIGNAL y : State_type ;
    SIGNAL RRin, Rin, Rout : STD_LOGIC_VECTOR(1 TO 3) ;
    SIGNAL R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL Extern : STD_LOGIC;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1' ) THEN
            CASE y IS
                WHEN A =>
                    IF w = '1' THEN
                        y <= B ;
                    ELSE
                        y <= A ;
                    END IF ;
                WHEN B =>
                    y <= C ;
                WHEN C =>
                    y <= D ;
                WHEN D =>
                    y <= A ;
            END CASE ;
        END IF ;
    END PROCESS ;

    … continued in Part b
```

Figure 7.7a A digital system like the one in Figure 7.2 (Part *a*).

```
    Extern <= '1' WHEN y = A ELSE '0' ;
    Rout(2) <= '1' WHEN y = B ELSE '0' ;
    RRin(3) <= '1' WHEN y = B ELSE '0' ;
    Rout(1) <= '1' WHEN y = C ELSE '0' ;
    RRin(2) <= '1' WHEN y = C ELSE '0' ;
    Rout(3) <= '1' WHEN y = D ELSE '0' ;
    RRin(1) <= '1' WHEN y = D ELSE '0' ;
    Done <= '1' WHEN y = D ELSE '0' ;

    Rin(1) <= RinExt(1) OR RRin(1) ;
    Rin(2) <= RinExt(2) OR RRin(2) ;
    Rin(3) <= RinExt(3) OR RRin(3) ;

    -- instantiate registers
    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    -- instantiate tri-state drivers
    tri_ext: trin PORT MAP ( Data, Extern, BusWires ) ;
    tri1: trin PORT MAP ( R1, Rout(1), BusWires ) ;
    tri2: trin PORT MAP ( R2, Rout(2), BusWires ) ;
    tri3: trin PORT MAP ( R3, Rout(3), BusWires ) ;
END Behavior ;
```

Figure 7.7b A digital system like the one in Figure 7.2 (Part *b*).

We assume that a three-bit input named *RinExt* exists, which allows the externally supplied data to be loaded from the bus into register *R*1, *R*2, or *R*3. This signal is not shown in Figure 7.2, to keep the figure simple, but it would be generated by the same external circuit block that produces *Data*. When *RinExt(1)* = 1, the data on the bus is loaded into register *R*1; when *RinExt(2)* = 1, the data is loaded into *R*2; and when *RinExt(3)* = 1, the data is loaded into *R*3.

The FSM in Figure 7.7*a* that implements the control circuit for our swapping task performs only one operation, hence we have not included the *Function* input from Figure 7.2. We have assumed that an input signal named *w* exists, which is set to 1 for one clock cycle to start the swapping task. We have also assumed that at the end of the swapping task, which is indicated by the *Done* signal being asserted, the control circuit returns to the starting state.

VHDL Code Using Multiplexers

Figure 7.8 shows how the code in Figure 7.7 can be modified to use multiplexers instead of tri-state drivers. Only the datapath circuit is shown, because the rest of the code is the same as that displayed in Figure 7.7*a*. Using the circuit structure shown in Figure 7.4, the bus is implemented with eight 4-to-1 multiplexers. Three of the data inputs on each 4-to-1 multiplexer are connected to one bit from registers *R*1, *R*2, and *R*3. The fourth data input is connected to one bit of the *Data* input signal to allow externally supplied data to be written into the

registers. In the VHDL code of Figure 7.8 the bus is described by using an IF-THEN-ELSE statement; other approaches could also be taken, such as using a CASE statement.

```
    Rout(2) <= '1' WHEN y = B ELSE '0' ;
    RRin(3) <= '1' WHEN y = B ELSE '0' ;
    Rout(1) <= '1' WHEN y = C ELSE '0' ;
    RRin(2) <= '1' WHEN y = C ELSE '0' ;
    Rout(3) <= '1' WHEN y = D ELSE '0' ;
    RRin(1) <= '1' WHEN y = D ELSE '0' ;
    Done <= '1' WHEN y = D ELSE '0' ;

    Rin(1) <= RinExt(1) OR RRin(1) ;
    Rin(2) <= RinExt(2) OR RRin(2) ;
    Rin(3) <= RinExt(3) OR RRin(3) ;

    – instantiate registers
    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    – define the bus
    PROCESS ( y, Data, R1, R2, R3 )
    BEGIN
        IF (y = B) THEN
            BusWires <= R2 ;
        ELSIF (y = C) THEN
            BusWires <= R1 ;
        ELSIF (y = D) THEN
            BusWires <= R3 ;
        ELSE
            BusWires <= Data ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 7.8 Using multiplexers to implement the bus structure.

Although the same FSM control circuit is used, the control signals *Extern* and *Rout* are not needed because tri-state drivers are not used. Instead, the required multiplexers are defined in the IF-THEN-ELSE statement by specifying the source of data based on the state of the FSM. When the FSM is in state *B*, the selected input to the multiplexers is *R*2. When the state is *C*, the register *R*1 provides the input data to the multiplexers, and so on.

7.2 Simple Processor

A second example of a digital system like the one in Figure 7.2 is shown in Figure 7.9. It has four *n*-bit registers, *R*0, …, *R*3, that are connected to the bus with tri-state drivers. External data can be loaded into the registers from the *n*-bit *Data* input, which is connected to the bus using tri-state drivers enabled by the *Extern* control signal. The system also includes an adder/subtractor module. One of its data inputs is provided by an *n*-bit register, *A*, that is attached to the bus, while the other data input, *B*, is directly connected to the bus. If the *AddSub* signal has the value 0, the module generates the sum *A*+*B*; if *AddSub* =1, the module generates the difference *A*-*B*. To perform the subtraction, we assume that the adder/subtractor includes the required XOR gates to form the 2's complement of *B*, as discussed in Section 3.3. The register *G* stores the output produced by the adder/subtractor. The *A* and *G* registers are controlled by the signals A_{in}, G_{in}, and G_{out}.

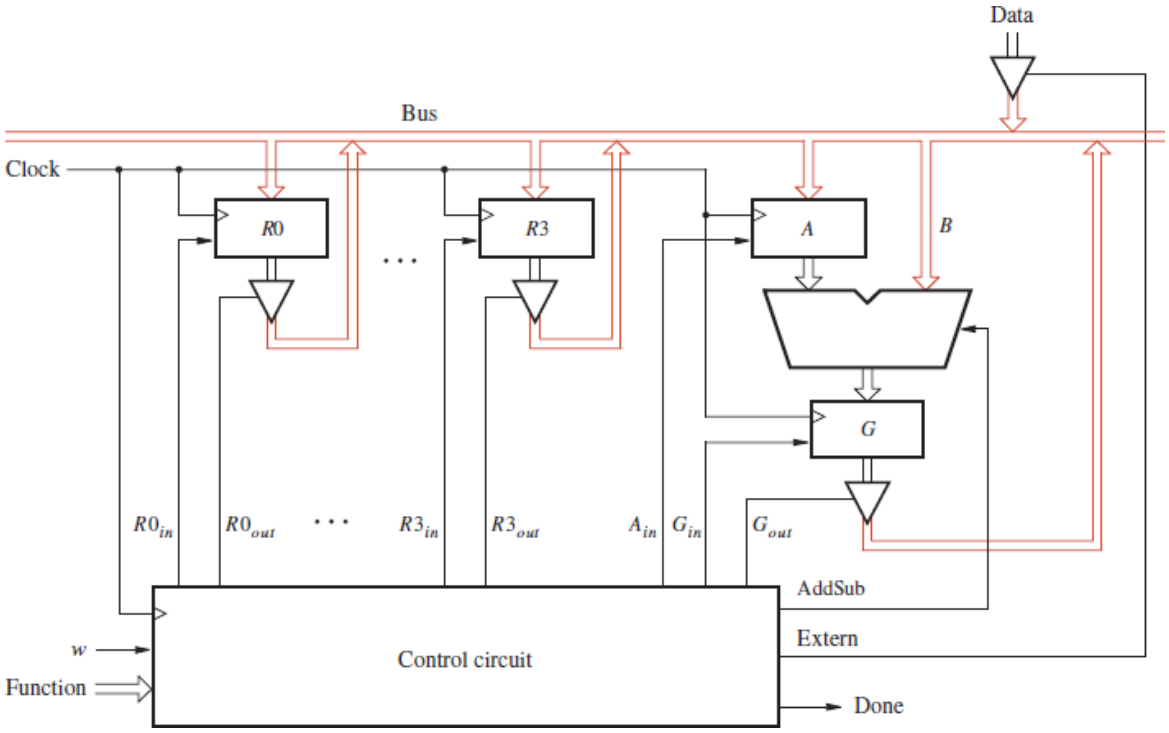


Figure 7.9 A digital system that implements a simple processor.

The system in Figure 7.9 can perform various functions, depending on the design of the control circuit. As an example, we will design a control circuit that can perform the four operations listed in Table 7.1. The left column in the table shows the name of an operation and its operands; the right column indicates the function performed in the operation. For the *Load* operation the meaning of *R*x ← *Data* is that the data on the external *Data* input is transferred across the bus into

any register, *Rx*, where *Rx* can be *R*0 to *R*3. The *Move* operation copies the data stored in register *Ry* into register *Rx*. In the table the square brackets, as in [*Rx*], refer to the *contents* of a register. Since only a single transfer across the bus is needed, both the *Load* and *Move* operations require only one step (clock cycle) to be completed. The *Add* and *Sub* operations require three steps, as follows: In the first step the contents of *Rx* are transferred across the bus into register *A*. Then in the next step, the contents of *Ry* are placed onto the bus. The adder/subtractor module performs the required function, and the results are stored in register *G*. Finally, in the third step the contents of *G* are transferred into *Rx*.

Table 7.1 Operations performed in the processor.

Operation	Function performed
Load *Rx*, *Data*	$Rx \leftarrow Data$
Move *Rx*, *Ry*	$Rx \leftarrow [Ry]$
Add *Rx*, *Ry*	$Rx \leftarrow [Rx] + [Ry]$
Sub *Rx*, *Ry*	$Rx \leftarrow [Rx] - [Ry]$

A digital system that performs the types of operations listed in Table 7.1 is usually called a *processor*. The specific operation to be performed at any given time is indicated using the control circuit input named *Function*. The operation is initiated by setting the *w* input to 1, and the control circuit asserts the *Done* output when the operation is completed.

The control circuit in Figure 7.9 can be implemented by using an FSM. But to illustrate another approach, we will base the design of the control circuit on a counter. This circuit has to generate the required control signals in each step of each operation. Since the longest operations (*Add* and *Sub*) need three steps (clock cycles), a two-bit counter can be used. Figure 7.10 shows a two-bit up-counter connected to a 2-to-4 decoder. Decoders are discussed in Chapter 4. The decoder is enabled at all times by setting its enable (*En*) input permanently to the value 1. Each of the decoder outputs represents a step in an operation. When no operation is currently being performed, the count value is 00; hence the T_0 output of the decoder is asserted. In the first step of an operation, the count value is 01, and T_1 is asserted. During the second and third steps of the *Add* and *Sub* operations, T_2 and T_3 are asserted, respectively.

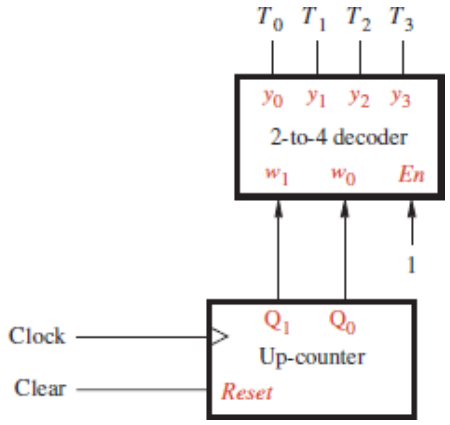


Figure 7.10 A part of the control circuit for the processor.

In each of steps T_0 to T_3, various control signal values have to be generated by the control circuit, depending on the operation being performed. Figure 7.11 shows that the operation is specified with six bits, which form the *Function* input. The two leftmost bits, $F=f_1$, f_0, are used as a two-bit number that identifies the operation. To represent *Load*, *Move*, *Add*, and *Sub*, we use the codes f_1, f_0 = 00, 01, 10, and 11, respectively. The inputs Rx_1 Rx_0 are a binary number that identifies the *Rx* operand, while Ry_1 Ry_0 identifies the *Ry* operand. The *Function* inputs are stored in a six-bit Function register when the FR_{in} signal is asserted.
0

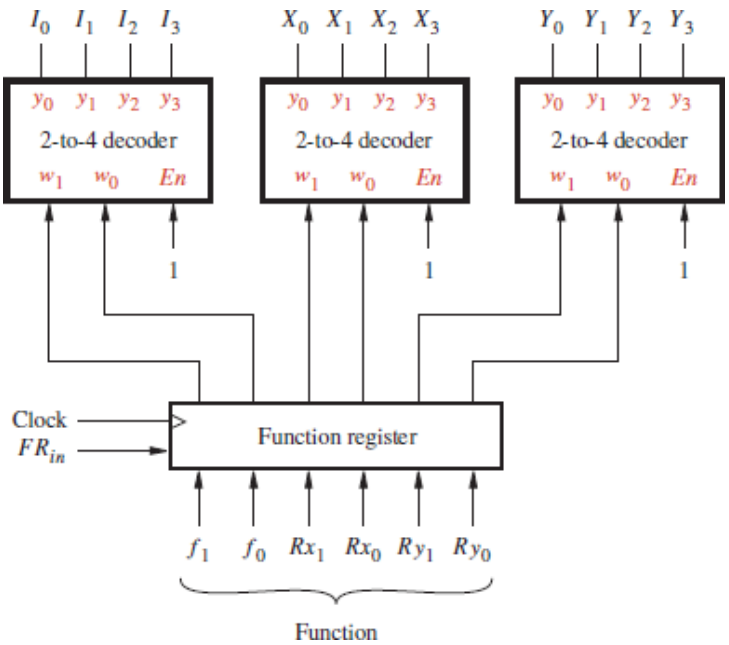


Figure 7.11 The function register and decoders.

Figure 7.11 also shows three 2-to-4 decoders that are used to decode the information encoded in the *F*, *Rx*, and *Ry* inputs. We will see shortly that these decoders are included as a convenience because their outputs provide simple-looking logic expressions for the various control signals.

The circuits in Figures 7.10 and 7.11 form a part of the control circuit. Using the input *w* and the signals T_0, …, T_3, I_0, …, I_3, X_0, …, X_3, and Y_0, …, Y_3, we will show how to derive the rest of the control circuit. It has to generate the outputs *Extern*, *Done*, A_{in}, G_{in}, G_{out}, *AddSub*, $R0_{in}$, …, $R3_{in}$, and $R0_{out}$, …, $R3_{out}$. The control circuit also has to generate the *Clear* and FR_{in} signals used in Figures 7.10 and 7.11.

Clear and FR_{in} are defined in the same way for all operations. *Clear* is used to ensure that the count value remains at 00 as long as *w*=0 and no operation is being executed. Also, it is used to clear the count value to 00 at the end of each operation. Hence an appropriate logic expression is

$$Clear = \overline{w}T_0 + Done$$

The FR_{in} signal is used to load the values on the *Function* inputs into the Function register when *w* changes to 1. Hence

$$FR_{in} = wT_0$$

The rest of the outputs from the control circuit depend on the specific step being performed in each operation. The values that have to be generated for each signal are shown in Table 7.2. Each row in the table corresponds to a specific operation, and each column represents one time step. The *Extern* signal is asserted only in the first step of the *Load* operation. Therefore, the logic expression that implements this signal is

Table 7.2 Control signals asserted in each operation/time step.

	T_1	T_2	T_3
(Load): I_0	$Extern, R_{in} = X,$ $Done$		
(Move): I_1	$R_{in} = X, R_{out} = Y,$ $Done$		
(Add): I_2	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in},$ $AddSub = 0$	$G_{out}, R_{in} = X,$ $Done$
(Sub): I_3	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in},$ $AddSub = 1$	$G_{out}, R_{in} = X,$ $Done$

$$Extern = I_0T_1$$

Done is asserted in the first step of *Load* and *Move*, as well as in the third step of *Add* and *Sub*. Hence

$$Done = (I_0 + I_1)T_1 + (I_2 + I_3)T_3$$

The A_{in}, G_{in}, and G_{out} signals are asserted in the *Add* and *Sub* operations. A_{in} is asserted in step T_1, G_{in} is asserted in T_2, and G_{out} is asserted in T_3. The *AddSub* signal has to be set to 0 in the *Add* operation and to 1 in the *Sub* operation. This is achieved with the following logic expressions

$$\begin{aligned} A_{in} &= (I_2 + I_3)T_1 \\ G_{in} &= (I_2 + I_3)T_2 \\ G_{out} &= (I_2 + I_3)T_3 \\ AddSub &= I_3 \end{aligned}$$

The values of $R0_{\text{in}}$, …, $R3_{in}$ are determined using either the X_0, …, X_3 signals or the Y_0, …, Y_3 signals. In Table 7.2 these actions are indicated by writing either $R_{\text{in}} = X$ or $R_{\text{in}} = Y$. The meaning of $R_{\text{in}} = X$ is that $R0_{\text{in}} = X_0$, $R1_{\text{in}} = X_1$, and so on. Similarly, the values of $R0_{\text{out}}$, …, $R3_{out}$ are specified using either $R_{\text{out}} = X$ or $R_{\text{out}} = Y$.

We will develop the expressions for $R0_{\text{in}}$ and $R0_{\text{out}}$ by examining Table 7.2 and then show how to derive the expressions for the other register control signals. The table shows that $R0_{in}$ is set to the value of X_0 in the first step of both the *Load* and *Move* operations and in the third step of both the *Add* and *Sub* operations, which leads to the expression

$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0$$

Similarly, $R0_{out}$ is set to the value of Y_0 in the first step of *Move*. It is set to X_0 in the first step of *Add* and *Sub* and to Y_0 in the second step of these operations, which gives

$$R0_{out} = I_1T_1Y_0 + (I_2 + I_3)(T_1X_0 + T_2Y_0)$$

The expressions for $R1_{in}$ and $R1_{out}$ are the same as those for $R0_{in}$ and $R0_{out}$ except that X_1 and Y_1 are used in place of X_0 and Y_0. The expressions for $R2_{in}$, $R2_{out}$, $R3_{in}$, and $R3_{out}$ are derived in the same way.

The circuits shown in Figures 7.10 and 7.11, combined with the circuits represented by the above expressions, implement the control circuit in Figure 7.9.

Processors are extremely useful circuits that are widely used. We have presented only the most basic aspects of processor design. However, the techniques presented can be extended to design realistic processors, such as modern microprocessors. The interested reader can refer to books on computer organization for more details on processor design.

VHDL Code

In this section we give two different styles of VHDL code for describing the system in Figure 7.9. The first style uses tri-state drivers to represent the bus, and it gives the logic expressions shown above for the outputs of the control circuit. The second style of code uses multiplexers to represent the bus, and it uses CASE statements that correspond to Table 7.2 to describe the outputs of the control circuit.

VHDL code for a two-bit up-counter is shown in Figure 7.12. It has a synchronous reset input, which is active high. For use with the processor, we created a VHDL package called *subccts*. The code is not shown here, but it includes component declarations for *upcount*, *regn* (Figure 7.5), *trin* (Figure 7.6), and *dec2to4* (Figure 4.26).

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY upcount IS
    PORT ( Clear, Clock  : IN      STD_LOGIC ;
           Q             : BUFFER  STD_LOGIC_VECTOR(1 DOWNTO 0) ) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
BEGIN
    upcount: PROCESS ( Clock )
    BEGIN
        IF (Clock'EVENT AND Clock = '1' ) THEN
            IF Clear = '1' THEN
                Q <= "00" ;
            ELSE
                Q <= Q + '1' ;
            END IF ;
        END IF;
    END PROCESS;
END Behavior ;
```

Figure 7.12 A two-bit up-counter with synchronous reset.

Complete code for the processor is given in Figure 7.13. In the architecture body, the statements labeled *counter* and *decT* instantiate the subcircuits in Figure

7.10. Note that we have assumed that the circuit has an active-high reset input, *Reset*, which is used to initialize the counter to 00. The statement Func < = *F* & *Rx* & *Ry* uses the concatenate operator to create the six-bit signal *Func*, which represents the inputs to the Function register in Figure 7.11. The next statement instantiates the Function register with the data inputs *Func* and the outputs *FuncReg*. The statements labeled *decI*, *decX*, and *decY* instantiate the decoders in Figure 7.11. Following these statements the previously derived logic expressions for the outputs of the control circuit are given. For $R0_{\text{in}}$, …, $R3_{in}$ and $R0_{out}$, …, $R3_{out}$, a GENERATE statement is used to produce the expressions.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;
USE work.subccts.all ;

ENTITY proc IS
    PORT ( Data      : IN     STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Reset, w  : IN     STD_LOGIC ;
           Clock     : IN     STD_LOGIC ;
           F, Rx, Ry : IN     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           Done      : BUFFER STD_LOGIC ;
           BusWires  : INOUT  STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END proc ;

ARCHITECTURE Behavior OF proc IS
    SIGNAL Rin, Rout : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL Clear, High, AddSub : STD_LOGIC ;
    SIGNAL Extern, Ain, Gin, Gout, FRin : STD_LOGIC ;
    SIGNAL Count, Zero : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
    SIGNAL T, I, X, Y : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL R0, R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL A, Sum, G : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL Func, FuncReg : STD_LOGIC_VECTOR(1 TO 6) ;
BEGIN
    Zero <= "00" ; High <= '1' ;
    Clear <= Reset OR Done OR (NOT w AND T(0)) ;
    counter: upcount PORT MAP ( Clear, Clock, Count ) ;
    decT: dec2to4 PORT MAP ( Count, High, T );
    Func <= F & Rx & Ry ;
    FRin <= w AND T(0) ;
    functionreg: regn GENERIC MAP ( N => 6 )
                 PORT MAP ( Func, FRin, Clock, FuncReg ) ;
    decI: dec2to4 PORT MAP ( FuncReg(1 TO 2), High, I ) ;
    decX: dec2to4 PORT MAP ( FuncReg(3 TO 4), High, X ) ;
    decY: dec2to4 PORT MAP ( FuncReg(5 TO 6), High, Y ) ;
    Extern <= I(0) AND T(1) ;
    Done <= ((I(0) OR I(1)) AND T(1)) OR ((I(2) OR I(3)) AND T(3)) ;
    Ain <= (I(2) OR I(3)) AND T(1) ;
    Gin <= (I(2) OR I(3)) AND T(2) ;
    Gout <= (I(2) OR I(3)) AND T(3) ;
    AddSub <= I(3) ;

... continued in Part b.
```

Figure 7.13 Code for the processor (Part *a*).

At the end of the code, the tri-state drivers and registers in the processor are instantiated, and the adder/subtractor module is described using a selected signal assignment.

```
    RegCntl:
    FOR k IN 0 TO 3 GENERATE
        Rin(k) <= ((I(0) OR I(1)) AND T(1) AND X(k)) OR
            ((I(2) OR I(3)) AND T(3) AND X(k)) ;
        Rout(k) <= (I(1) AND T(1) AND Y(k)) OR
            ((I(2) OR I(3)) AND ((T(1) AND X(k)) OR (T(2) AND Y(k)))) ;
    END GENERATE RegCntl ;
    tri_extern: trin PORT MAP ( Data, Extern, BusWires ) ;
    reg0: regn PORT MAP ( BusWires, Rin(0), Clock, R0 ) ;
    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    tri0: trin PORT MAP ( R0, Rout(0), BusWires ) ;
    tri1: trin PORT MAP ( R1, Rout(1), BusWires ) ;
    tri2: trin PORT MAP ( R2, Rout(2), BusWires ) ;
    tri3: trin PORT MAP ( R3, Rout(3), BusWires ) ;
    regA: regn PORT MAP ( BusWires, Ain, Clock, A ) ;
    alu:
    WITH AddSub SELECT
        Sum <= A + BusWires WHEN '0' ,
               A - BusWires WHEN OTHERS ;
    regG: regn PORT MAP ( Sum, Gin, Clock, G ) ;
    triG: trin PORT MAP ( G, Gout, BusWires ) ;
END Behavior ;
```

Figure 7.13 Code for the processor (Part *b*).

Using Multiplexers and Case Statements

We showed in Figure 7.4 that a bus can be implemented using multiplexers, rather than tri-state drivers. VHDL code that describes the processor using this approach is shown in Figure 7.14. The same entity declaration given in Figure 7.13 can be used and is not shown in Figure 7.14. The code illustrates a different way of describing the control circuit in the processor. It does not give logic expressions for the signals *Extern*, *Done*, and so on, as we did in Figure 7.13. Instead, CASE statements are used to represent the information shown in Table 7.2. These statements are provided inside the process labeled *controlsignals*. Each control signal is first assigned the value 0, as a default. This is required because the CASE statements specify the values of the control signals only when they should be asserted, as we did in Table 7.2. As explained for Figure 4.39, when the value of a signal is not specified, the signal retains its current value. This implied memory results in a feedback connection in the synthesized circuit. We avoid this problem by providing the default value of 0 for each of the control signals involved in the CASE statements.

```
ARCHITECTURE Behavior OF proc IS
    SIGNAL X, Y, Rin, Rout : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL Clear, High, AddSub : STD_LOGIC ;
    SIGNAL Extern, Ain, Gin, Gout, FRin : STD_LOGIC ;
    SIGNAL Count, Zero, T, I : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
    SIGNAL R0, R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL A, Sum, G : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL Func, FuncReg, Sel : STD_LOGIC_VECTOR(1 TO 6) ;
BEGIN
    Zero <= "00" ; High <= '1' ;
    Clear <= Reset OR Done OR (NOT w AND NOT T(1) AND NOT T(0)) ;
    counter: upcount PORT MAP ( Clear, Clock, Count ) ;
    T <= Count ;
    Func <= F & Rx & Ry ;
    FRin <= w AND NOT T(1) AND NOT T(0) ;
    functionreg: regn GENERIC MAP ( N => 6 )
                    PORT MAP ( Func, FRin, Clock, FuncReg ) ;
    I <= FuncReg(1 TO 2) ;
    decX: dec2to4 PORT MAP ( FuncReg(3 TO 4), High, X ) ;
    decY: dec2to4 PORT MAP ( FuncReg(5 TO 6), High, Y ) ;

    controlsignals: PROCESS ( T, I, X, Y )
    BEGIN
        Extern <= '0' ; Done <= '0' ; Ain <= '0' ; Gin <= '0' ;
        Gout <= '0' ; AddSub <= '0' ; Rin <= "0000" ; Rout <= "0000" ;
        CASE T IS       WHEN "00" => -- no signals asserted in time step T0
            WHEN "01" => -- define signals asserted in time step T1
                CASE I IS
                    WHEN "00" => -- Load
                        Extern <= '1' ; Rin <= X ; Done <= '1' ;
                    WHEN "01" => -- Move
                        Rout <= Y ; Rin <= X ; Done <= '1' ;
                    WHEN OTHERS => -- Add, Sub
                        Rout <= X ; Ain <= '1' ;
                END CASE ;
```

… continued in Part *b*

Figure 7.14 Alternative code for the processor (Part *a*).

```
            WHEN "10" => -- define signals asserted in time step T2
                CASE I IS
                    WHEN "10" => -- Add
                        Rout <= Y ; Gin <= '1' ;
                    WHEN "11" => -- Sub
                        Rout <= Y ; AddSub <= '1' ; Gin <= '1' ;
                    WHEN OTHERS => -- Load, Move
                END CASE ;
            WHEN OTHERS => -- define signals asserted in time step T3
                CASE I IS
                    WHEN "00" => -- Load
                    WHEN "01" => -- Move
                    WHEN OTHERS => -- Add, Sub
                        Gout <= '1' ; Rin <= X ; Done <= '1' ;
                END CASE ;
        END CASE ;
    END PROCESS ;
    reg0: regn PORT MAP ( BusWires, Rin(0), Clock, R0 ) ;
    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    regA: regn PORT MAP ( BusWires, Ain, Clock, A ) ;
    alu: WITH AddSub SELECT
        Sum <= A + BusWires WHEN '0' ,
                A - BusWires WHEN OTHERS ;
    regG: regn PORT MAP ( Sum, Gin, Clock, G ) ;
    Sel <= Rout & Gout & Extern ;
    WITH Sel SELECT
        BusWires <= R0 WHEN "100000" ,
                    R1 WHEN "010000" ,
                    R2 WHEN "001000" ,
                    R3 WHEN "000100" ,
                    G WHEN  "000010" ,
                    Data WHEN OTHERS ;
END Behavior ;
```

Figure 7.14 Alternative code for the processor (Part *b*).

In Figure 7.13 the statements labeled *decT* and *decI* are used to decode the *Count* signal and the stored values of the *F* input, respectively. The *decT* decoder has the outputs T_0, …, T_3, and *decI* produces I_0, …, I_3. In Figure 7.14 these two decoders are not used, because they do not serve a useful purpose in this code. Instead, the signals *T* and *I* are defined as two-bit signals, which are used in the CASE statements. The code sets *T* to the value of *Count*, while *I* is set to the value of the two leftmost bits in the Function Register, which correspond to the stored values of the input *F*.

There are two nested levels of CASE statements. The first one enumerates the possible values of *T*. For each WHEN clause in this CASE statement, which represents a column in Table 7.2, there is a nested CASE statement that

enumerates the four values of *I*. As indicated by the comments in the code, the nested CASE statements correspond exactly to the information given in Table 7.2.

At the end of Figure 7.14, the bus is described using a selected signal assignment. This statement represents multiplexers that place the appropriate data onto *BusWires*, depending on the values of R_{out}, G_{out}, and *Extern*.

The circuits synthesized from the code in Figures 7.13 and 7.14 are functionally equivalent. The style of code in Figure 7.14 has the advantage that it does not require the manual effort of analyzing Table 7.2 to generate the logic expressions for the control signals used for Figure 7.13. By using the style of code in Figure 7.14, these expressions are produced automatically by the VHDL compiler as a result of analyzing the CASE statements. The style of code in Figure 7.14 is less prone to careless errors. Also, using this style of code it would be straightforward to provide additional capabilities in the processor, such as adding other operations.

Figure 7.15 gives an example of the results of a simulation of the code in Figure 7.14. Each clock cycle in which *w*=1 in the simulation waveforms indicates the start of an operation. In the first such operation, at 250 ns in the simulation time, the values of both inputs *F* and *Rx* are 00. Hence the operation corresponds to "*Load R*0, *Data*." The value of *Data* is 2A, which is loaded into *R*0 on the next positive clock edge. The next operation loads 55 into register *R*1, and the subsequent operation loads 22 into *R*2. At 850 ns the value of the input *F* is 10, while *Rx* = 01 and *Ry* = 00. This operation is "*Add R*1, *R*0." In the following clock cycle, the contents of *R*1 (55) appear on the bus. This data is loaded into register *A* by the clock edge at 950 ns, which also results in the contents of *R*0 (2A) being placed on the bus. The adder/subtractor module generates the correct sum (7F), which is loaded into register *G* at 1050 ns. After this clock edge the new contents of *G* (7F) are placed on the bus and loaded into register *R*1 at 1150 ns. Two more operations are shown in the timing diagram. The one at 1250 ns ("*Move R*3, *R*1") copies the contents of *R*1 (7F) into *R*3. Finally, the operation starting at 1450 ns ("*Sub R*3, *R*2") subtracts the contents of *R*2 (22) from the contents of *R*3 (7F), producing the correct result, 7F - 22 = 5D.

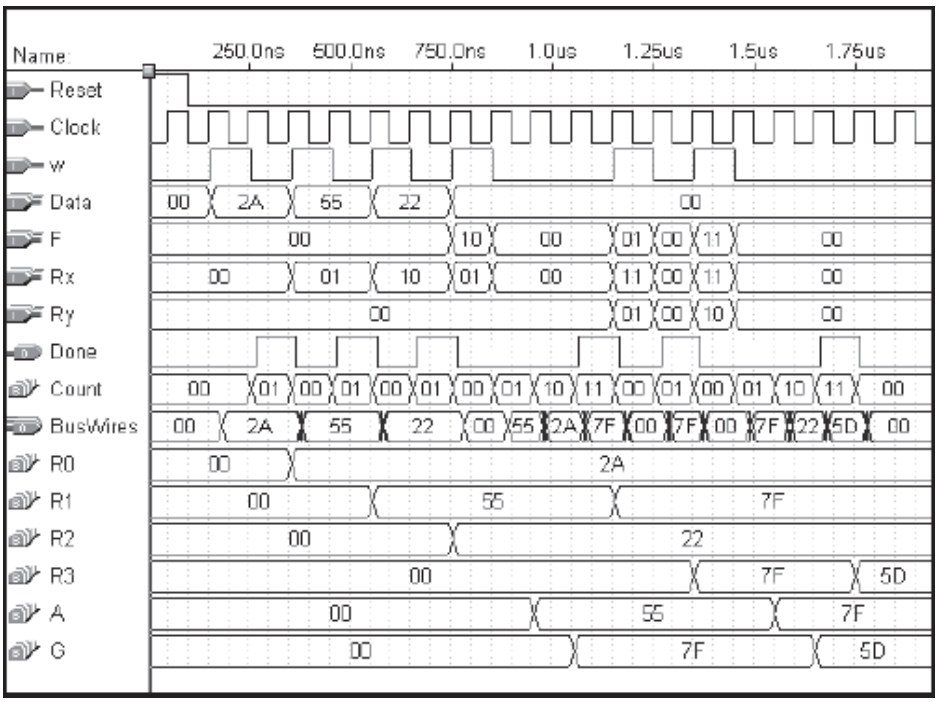


Figure 7.15 Simulation for the VHDL code in Figure 7.14.

7.3 A Bit-Counting Circuit

We introduced algorithmic state machine (ASM) charts in Section 6.10 and showed how they can be used to describe finite state machines. ASM charts can also be used to describe digital systems that include both datapath and control circuits. We will illustrate how the ASM charts can be used as an aid in designing digital systems by using them in the remaining examples in this chapter.

Suppose that we wish to count the number of bits in a register, *A*, that have the value 1. Figure 7.16 shows pseudo-code for a step-by-step procedure, or *algorithm*, that can be used to perform the required task. It assumes that *A* is stored in a register that can shift its contents in the left-to-right direction. The answer produced by the algorithm is stored in the variable named *B*. The algorithm terminates when *A* does not contain any more 1s, that is when $A = 0$. In each iteration of the while loop, if the least-significant bit (LSB) of *A* is 1, then B is incremented by 1; otherwise, *B* is not changed. The contents of *A* are shifted one bit to the right at the end of each loop iteration.

```
B = 0;
while A ≠ 0 do
    if a0 = 1 then
        B = B + 1;
    end if;
    Right-shift A;
end while;
```

Figure 7.16 Pseudo-code for the bit counter.

Figure 7.17 gives an ASM chart that represents the algorithm in Figure 7.16. The state box for the starting state, *S*1, specifies that *B* is initialized to 0. We assume that an input signal, *s*, exists, which is used to indicate when the data to be processed has been loaded into *A*, so that the machine can start. The decision box labeled *s* stipulates that the machine remains in state *S*1 as long as *s*=0. The conditional output box with *Load A* written inside it indicates that *A* is loaded from external data inputs if *s*=0 in state *S*1.

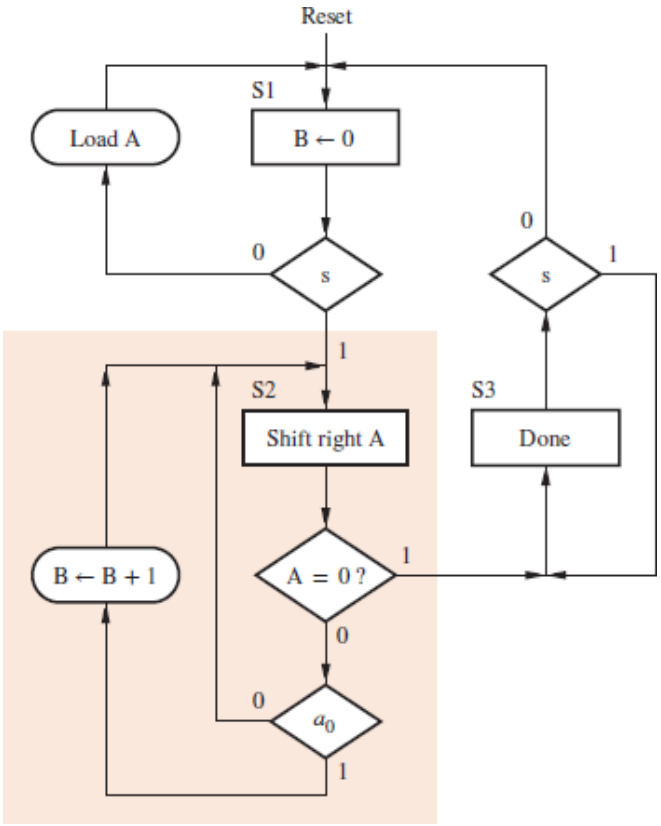


Figure 7.17 ASM chart for the pseudo-code in Figure 7.16.

When *s* becomes 1, the machine changes to state *S*2. The decision box below the state box for *S*2 checks whether *A*=0. If so, the bit-counting operation is complete; hence the machine should change to state *S*3. If not, the FSM remains

in state $S2$. The decision box at the bottom of the chart checks the value of a_0. If $a_0 = 1$, B is incremented, which is indicated in the chart as $B \leftarrow B + 1$. If $a_0=0$, then B is not changed. In state $S3$, B contains the result, which is the number of bits in A that were 1. An output signal, *Done*, is set to 1 to indicate that the algorithm is finished; the FSM stays in $S3$ until s goes back to 0.

ASM Chart Implied Timing Information

In Chapter 6 we said that ASM charts are similar to traditional flowcharts, except that the ASM chart implies timing information. We can use the bit-counting example to illustrate this concept. Consider the ASM block for state $S2$, which is shaded in red in Figure 7.17. In a traditional flowchart, when state $S2$ is entered, the value of A would first be shifted to the right. Then we would examine the value of A and if A's LSB is 1, we would immediately add 1 to B. But, since the ASM chart represents a sequential circuit, changes in A and B, which represent the outputs of flip-flops, take place after the active clock edge. The same clock signal that controls changes in the state of the machine also controls changes in A and B. When the machine is in state $S1$, the next active clock edge will only perform the action specified inside the state box for $S1$, which is $B \leftarrow 0$. Hence in state $S2$, the decision box that tests whether $A=0$, as well as the box that checks the value of a_0, check the bits in A before they are shifted. If $A = 0$, then the FSM will change to state $S3$ on the next clock edge (this clock edge also shifts A, which has no effect because A is already 0 in this case.) On the other hand, if $A \neq 0$, then the FSM does not change to $S3$, but remains in $S2$. At the same time, A is still shifted, and B is incremented if a_0 has the value 1. These timing issues are illustrated in Figure 7.21, which represents a simulation result for a circuit that implements the ASM chart. We show how the circuit is designed in the following discussion.

Datapath Circuit

By examining the ASM chart for the bit-counting circuit, we can infer the type of circuit elements needed to implement its datapath. We need a shift register that shifts left-to-right to implement A. It must have the parallel-load capability because of the conditional output box in state $S1$ that loads data into the register. An enable input is also required because shifting should occur only in state $S2$. A counter is needed for B, and it needs a parallel-load capability to initialize the count to 0 in state $S1$. It is not wise to rely on the counter's reset input to clear B to 0 in state $S1$. In practice, the reset signal is used in a digital system for only two purposes: to initialize the circuit when power is first applied, or to recover from

an error. The machine changes from state $S3$ to $S1$ as a result of s=0; hence we should not assume that the reset signal is used to clear the counter.

The datapath circuit is depicted in Figure 7.18. The serial input to the shift register, w, is connected to 0, because it is not needed. The load and enable inputs on the shift register are driven by the signals LA and EA. The parallel input to the shift register is named $Data$, and its parallel output is A. An n-input NOR gate is used to test whether A=0. The output of this gate, z, is 1 when A=0. Note that the figure indicates the n-input NOR gate by showing a single input connection to the gate, with the label n attached to it. The counter has $log_2(n)$ bits, with parallel inputs connected to 0 and parallel outputs named B. It also has a parallel-load input LB and enable input EB control signals.

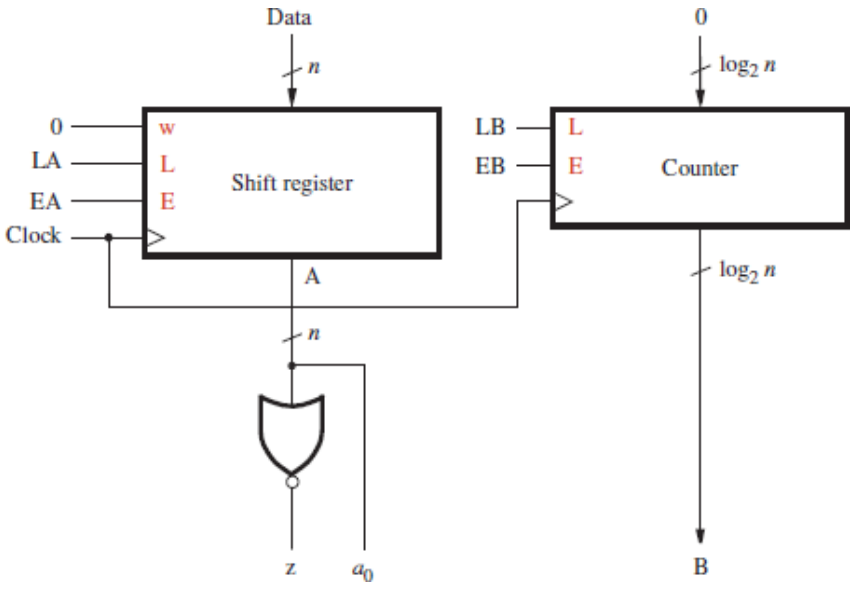


Figure 7.18 Datapath for the ASM chart in Figure 7.17.

Control Circuit

For convenience we can draw a second ASM chart that represents only the FSM needed for the control circuit, as shown in Figure 7.19. The FSM has the inputs s, a_0, and z and generates the outputs EA, LB, EB, and $Done$. In state $S1$, LB is asserted, so that 0 is loaded in parallel into the counter. Note that for the control signals, like LB, instead of writing LB =1, we simply write LB to indicate that the signal is asserted. We assume that external circuitry drives LA to 1 when valid data is present at the parallel inputs of the shift register, so that the shift register contents are initialized before s changes to 1. In state $S2$, EA is asserted to cause a shift operation, and the count enable for B is asserted only if a_0=1.

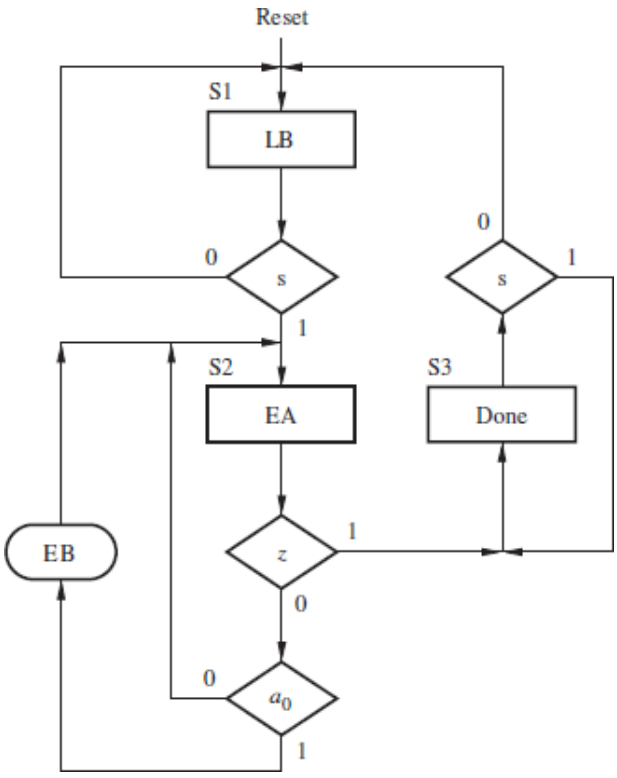


Figure 7.19 ASM chart for the bit counter control circuit.

VHDL Code

The bit-counting circuit can be described in VHDL code as shown in Figure 7.20. We have chosen to define *A* as an eight-bit STD_LOGIC_VECTOR signal and *B* as an integer signal. The ASM chart in Figure 7.19 can be directly translated into code that describes the required control circuit. The signal named *y* is used to represent the state flip-flops, and the process labeled *FSM_transitions*, at the top of the architecture body, specifies the state transitions. The process labeled *FSM_outputs* specifies the generated outputs in each state. A default value is specified at the beginning of this process for all output signals, and then individual output values are specified in the case statement.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY work ;
USE work.components.shiftrne ;

ENTITY bitcount IS
    PORT( Clock, Resetn  : IN      STD_LOGIC ;
          LA, s          : IN      STD_LOGIC ;
          Data           : IN      STD_LOGIC_VECTOR(7 DOWNTO 0) ;
          B              : BUFFER  INTEGER RANGE 0 to 8 ;
          Done           : OUT     STD_LOGIC ) ;
END bitcount ;

ARCHITECTURE Behavior OF bitcount IS
    TYPE State_type IS ( S1, S2, S3 ) ;
    SIGNAL y : State_type ;
    SIGNAL A : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL z, EA, LB, EB, low : STD_LOGIC ;
BEGIN
    FSM_transitions: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= S1 ;
        ELSIF (Clock'EVENT AND Clock = '1' ) THEN
            CASE y IS
                WHEN S1 =>
                    IF s = '0' THEN y <= S1 ; ELSE y <= S2 ; END IF ;
                WHEN S2 =>
                    IF z = '0' THEN y <= S2 ; ELSE y <= S3 ; END IF ;
                WHEN S3 =>
                    IF s = '1' THEN y <= S3 ; ELSE y <= S1 ; END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
```

… continued in Part *b*

Figure 7.20 VHDL code for the bit-counting circuit (Part *a*).

```
    FSM_outputs: PROCESS ( y, s, A(0), z )
    BEGIN
        EA <= '0' ; LB <= '0' ; EB <= '0' ; Done <= '0' ;
        CASE y IS
            WHEN S1 =>
                LB <= '1' ;
            WHEN S2 =>
                EA <= '1' ;
                IF A(0) = '1' THEN EB <= '1' ; ELSE EB <= '0' ; END IF ;
            WHEN S3 =>
                Done <= '1' ;
        END CASE ;
    END PROCESS ;

    -- The datapath circuit is described below
    upcount: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            B <= 0 ;
        ELSIF (Clock'EVENT AND Clock = '1' ) THEN
            IF LB = '1' THEN
                B <= 0 ;
            ELSIF EB = '1' THEN
                B <= B + 1 ;
            END IF ;
        END IF;
    END PROCESS;

    low <= '0' ;
    ShiftA: shiftrne GENERIC MAP ( N => 8 )
        PORT MAP ( Data, LA, EA, low, Clock, A ) ;
    z <= '1' WHEN A = "00000000" ELSE '0' ;
END Behavior ;
```

Figure 7.20b VHDL code for the bit-counting circuit (Part *b*).

The process labeled *upcount* defines the up-counter that implements *B*. The shift register for *A* is instantiated at the end of the code, and the *z* signal is defined using a conditional signal assignment.

We performed a simulation of the code in Figure 7.20. Figure 7.21 gives the results of the simulation for *A*=00111011. After the circuit is reset, the input signal *LA* is set to 1, and the desired data, $(3B)_{16}$, is placed on the *Data* input. When *s* changes to 1, the next active clock edge causes the FSM to change to state *S*2. In this state each active clock edge increments *B* if a_0 is 1, and shifts *A*. When *A*=0, the next clock edge causes the FSM to change to state *S*3, where *Done* is set to 1 and *B* has the correct result, *B*=5. To check more thoroughly that the circuit is designed correctly, we should try different values of input data.

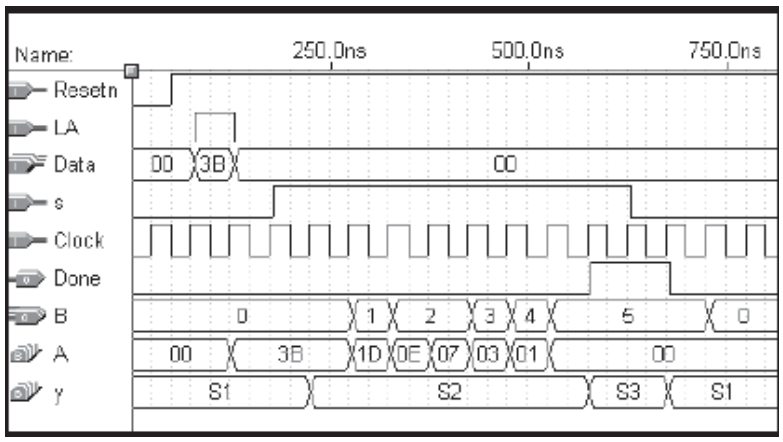


Figure 7.21 Simulation results for the bit-counting circuit.

7.4 Shift-and-Add Multiplier

We presented a circuit that multiplies two unsigned binary numbers in Figure 3.28. The circuit uses a two-dimensional array of identical subcircuits, each of which contains a full-adder and an AND gate. For larger *n*-bit numbers, this approach may not be appropriate because of the large number of gates needed. Another approach is to use a shift register in combination with an adder to implement the traditional method of multiplication that is done by "hand." Figure 7.22*a* illustrates the manual process of multiplying two binary numbers. The product is formed by a series of addition operations. For each bit i in the multiplier that is 1, we add to the product the value of the multiplicand shifted to the left i times. This algorithm can be described in pseudo-code as shown in Figure 7.22*b*, where A is the multiplicand, B is the multiplier, and P is the product.

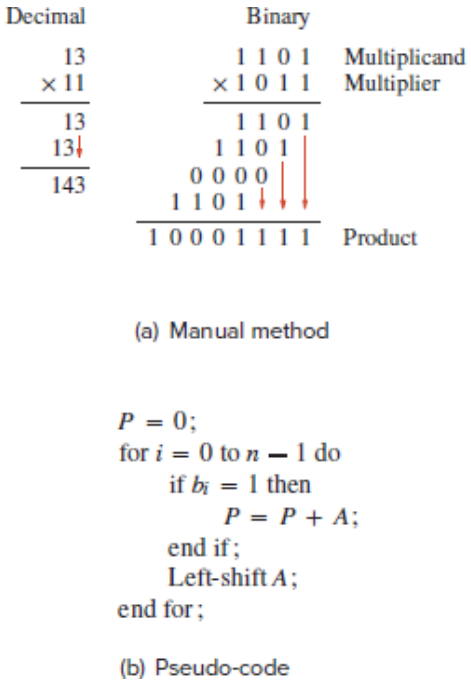


Figure 7.22 An algorithm for multiplication.

An ASM chart that represents the algorithm in Figure 7.22*b* is given in Figure 7.23. We assume that an input *s* is used to control when the machine begins the multiplication process. As long as *s* is 0, the machine stays in state *S*1 and the data for *A* and *B* can be loaded from external inputs. In state *S*2 we test the value of the LSB of *B*, and if it is 1, we add *A* to *P*. Otherwise, *P* is not changed. The machine moves to state *S*3 when *B* contains 0, because *P* has the final product in this case. For each clock cycle in which the machine is in state *S*2, we shift the value of *A* to the left, as specified in the pseudo-code in Figure 7.22*b*. We shift the contents of *B* to the right so that in each clock cycle b_0 can be used to decide whether or not *A* should be added to *P*.

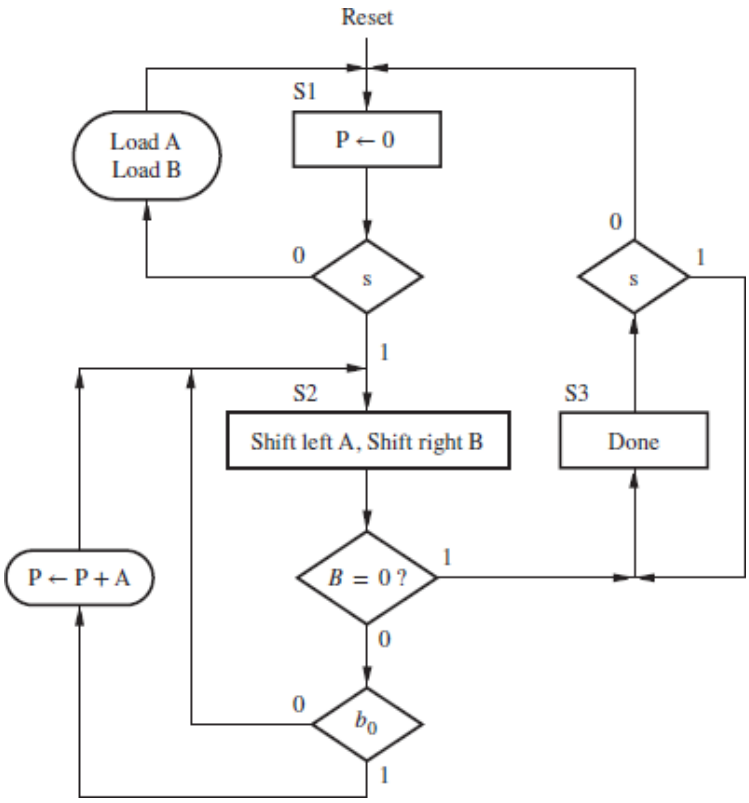


Figure 7.23 ASM chart for the multiplier.

Datapath Circuit

We can now define the datapath circuit. To implement *A* we need a right-to-left shift register that has 2*n* bits. A 2*n*-bit register is needed for *P*, and it must have an enable input because the assignment $P \leftarrow P + A$ in state *S*2 is inside a conditional output box. A 2*n*-bit adder is needed to produce *P* + *A*. Note that *P* is loaded with 0 in state *S*1, and *P* is loaded from the output of the adder in state *S*2. We cannot assume that the reset input is used to clear *P*, because the machine changes from state *S*3 back to *S*1 based on the *s* input, not the reset input. Hence a 2-to-1 multiplexer is needed for each input to *P*, to select either 0 or the appropriate sum bit from the adder. An *n*-bit left-to-right shift register is needed for *B*, and an *n*-input NOR gate can be used to test whether *B*=0.

Figure 7.24 shows the datapath circuit and labels the control signals for the shift registers. The input data for the shift register that holds *A* is named *DataA*. Since the shift register has 2*n* bits, the most-significant *n* data inputs are connected to 0. A single multiplexer symbol is shown connected to the register that holds *P*. This symbol represents 2*n* 2-to-1 multiplexers that are each controlled by the *Psel* signal.

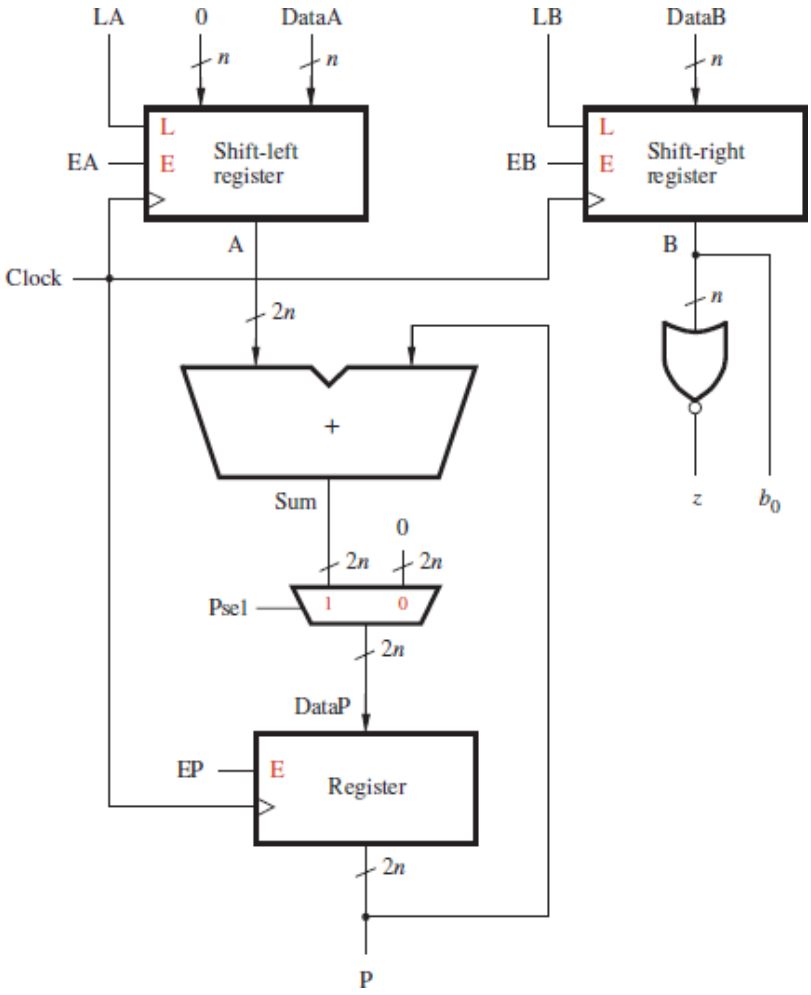


Figure 7.24 Datapath circuit for the multiplier.

Control Circuit

An ASM chart that represents only the control signals needed for the multiplier is given in Figure 7.25. In state $S1$, *Psel* is set to 0 and *EP* is asserted, so that register *P* is cleared. When s=0, parallel data can be loaded into shift registers *A* and *B* by an external circuit that controls their parallel-load inputs *LA* and *LB*. When s=1, the machine changes to state $S2$, where *Psel* is set to 1 and shifting of *A* and *B* is enabled. If b_0 = 1, the enable for *P* is asserted. The machine changes to state $S3$ when z=1, and then remains in $S3$ and sets *Done* to the value 1 as long as s=1.

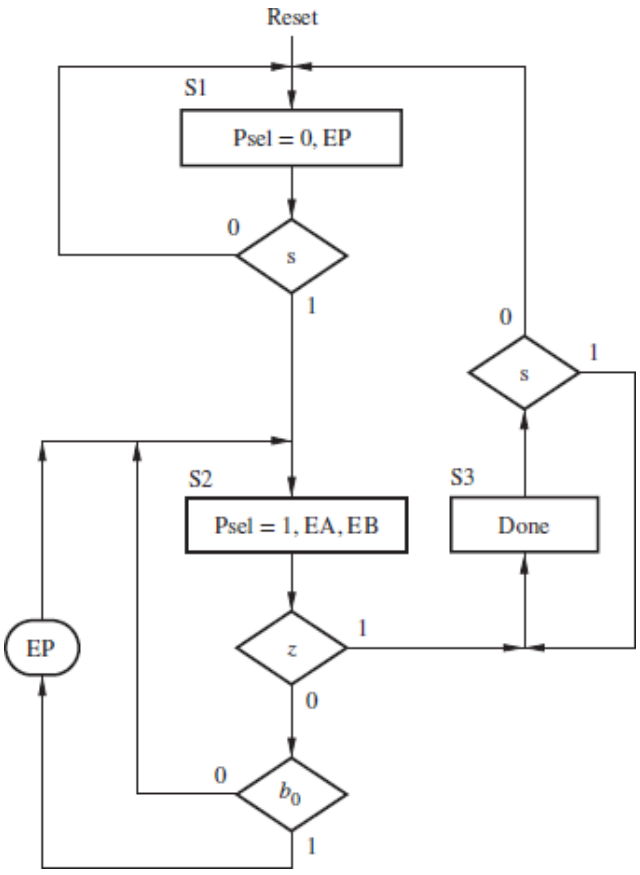


Figure 7.25 ASM chart for the multiplier control circuit.

VHDL Code

VHDL code for the multiplier is given in Figure 7.26. The number of bits in *A* and *B* is set by the generic parameter *N*. Since some registers are 2*n* bits wide, a second generic parameter *NN* is defined to represent 2 × *N*. By changing the value of the generic parameters, the code can be used for numbers of any size. The processes labeled *FSM_transitions* and *FSM_outputs* define the state transitions and generated outputs, respectively, in the control circuit. The parallel data input on the shift register *A* is 2*N* bits wide, but *DataA* is only *N* bits wide. The signal *N_Zeros* is used to generate *n* zero bits, and the signal *Ain* prepends these bits with *DataA* for loading into the shift register. The multiplexer needed for register *P* is defined using a FOR GENERATE statement that instantiates 2*N* 2-to-1 multiplexers. Figure 7.27 gives a simulation result for the circuit generated from the code. After the circuit is reset, *LA* and *LB* are set to 1, and the numbers to be multiplied are placed on the *DataA* and *DataB* inputs. After *s* is set to 1, the FSM (*y*) changes to state *S*2, where it remains until *B*=0. For each clock cycle in state *S*2, *A* is shifted to the left, and *B* is shifted to the right. In three of the clock cycles in state *S*2, the contents of *A* are added to *P*, corresponding to the three bits in *B*

that have the value 1. When B=0, the FSM changes to state $S3$ and P contains the correct product, which is $(64)_{16} \times (19)_{16} = (9C4)_{16}$. The decimal equivalent of this result is 100 × 25 = 2500.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;
USE work.components.all ;

ENTITY multiply IS
    GENERIC ( N : INTEGER := 8; NN : INTEGER := 16 ) ;
    PORT ( Clock       : IN      STD_LOGIC ;
           Resetn      : IN      STD_LOGIC ;
           LA, LB, s   : IN      STD_LOGIC ;
           DataA       : IN      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           DataB       : IN      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           P           : BUFFER  STD_LOGIC_VECTOR(NN-1 DOWNTO 0) ;
           Done        : OUT     STD_LOGIC ) ;
END multiply ;

ARCHITECTURE Behavior OF multiply IS
    TYPE State_type IS ( S1, S2, S3 ) ;
    SIGNAL y : State_type ;
    SIGNAL Psel, z, EA, EB, EP, Zero : STD_LOGIC ;
    SIGNAL B, N_Zeros : STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
    SIGNAL A, Ain, DataP, Sum : STD_LOGIC_VECTOR(NN-1 DOWNTO 0) ;
BEGIN
    FSM_transitions: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= S1 ;
        ELSIF (Clock'EVENT AND Clock = '1' ) THEN
            CASE y IS
                WHEN S1 =>
                    IF s = '0' THEN y <= S1 ; ELSE y <= S2 ; END IF ;
                WHEN S2 =>
                    IF z = '0' THEN y <= S2 ; ELSE y <= S3 ; END IF ;
                WHEN S3 =>
                    IF s = '1' THEN y <= S3 ; ELSE y <= S1 ; END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

... continued in Part b
```

Figure 7.26 VHDL code for the multiplier circuit (Part *a*).

```
    FSM_outputs: PROCESS ( y, s, LA, LB, B(0) )
    BEGIN
        EP <= '0' ; EA <= '0' ; EB <= '0' ; Done <= '0' ; Psel <= '0' ;
        CASE y IS
            WHEN S1 =>
                EP <= '1' ;
            WHEN S2 =>
                EA <= '1' ; EB <= '1' ; Psel <= '1' ;
                IF B(0) = '1' THEN EP <= '1' ; ELSE EP <= '0' ; END IF ;
            WHEN S3 =>
                Done <= '1' ;
        END CASE ;
    END PROCESS ;

    -- Define the datapath circuit
    Zero <= '0' ;
    N_Zeros <= (OTHERS => '0' ) ;
    Ain <= N_Zeros & DataA ;
    ShiftA: shiftlne GENERIC MAP ( N => NN )
        PORT MAP ( Ain, LA, EA, Zero, Clock, A ) ;
    ShiftB: shiftrne GENERIC MAP ( N => N )
        PORT MAP ( DataB, LB, EB, Zero, Clock, B ) ;
    z <= '1' WHEN B = N_Zeros ELSE '0' ;
    Sum <= A + P ;
    -- Define the 2n 2-to-1 multiplexers for DataP
    GenMUX: FOR i IN 0 TO NN-1 GENERATE
        Muxi: mux2to1 PORT MAP ( Zero, Sum(i), Psel, DataP(i) ) ;
    END GENERATE;
    RegP: regne GENERIC MAP ( N => NN )
        PORT MAP ( DataP, Resetn, EP, Clock, P ) ;
END Behavior ;
```

Figure 7.26b VHDL code for the multiplier circuit (Part *b*).

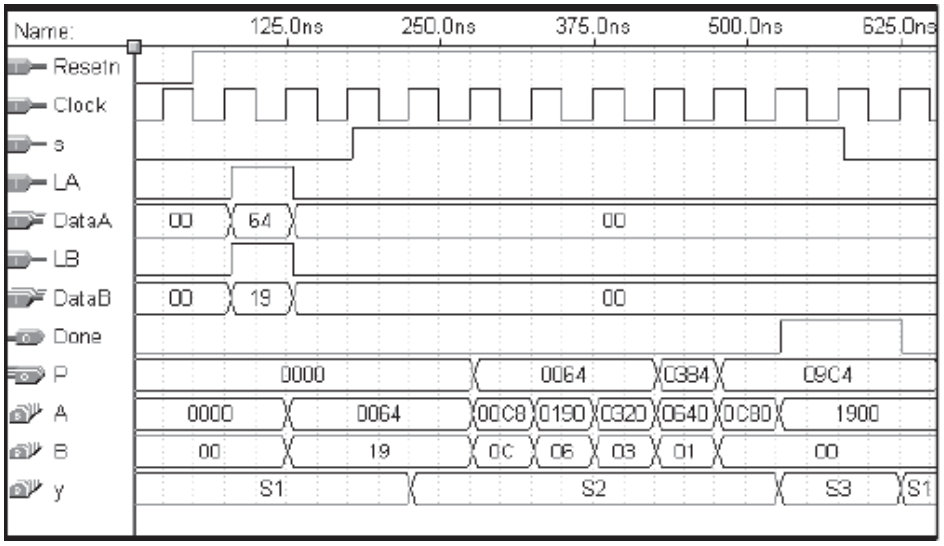
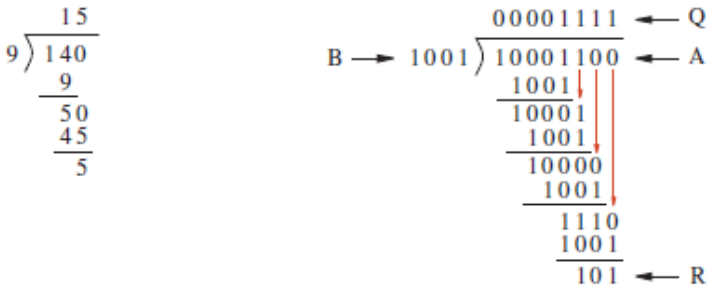


Figure 7.27 Simulation results for the multiplier circuit.

The number of clock cycles that the circuit requires to generate the final product is determined by the leftmost digit in *B* that is 1. It is possible to reduce the number of clock cycles needed by using more complex shift registers for *A* and *B*. If the two rightmost bits in *B* are both 0, then both *A* and *B* could be shifted by two bit positions in one clock cycle. Similarly, if the three lowest digits in *B* are 0, then a three bit-position shift can be done, and so on. A shift register that can shift by multiple bit positions at once can be built using a *barrel shifter*. We leave it as an exercise for the reader to modify the multiplier to make use of a barrel shifter.

7.5 DIVIDER

The preceding example implements the traditional method of performing multiplication by hand. In this example we will design a circuit that implements the traditional long-hand division. Figure 7.28*a* gives an example of long-hand division. The first step is to try to divide the divisor 9 into the first digit of the dividend 1, which does not work. Next, we try to divide 9 into 14, and determine that 1 is the first digit in the quotient. We perform the subtraction 14-9=5, bring down the last digit from the dividend to form 50, and then determine that the next digit in the quotient is 5. The remainder is 50-45 =5, and the quotient is 15. Using binary numbers, as illustrated in Figure 7.28*b*, involves the same process, with the simplification that each digit of the quotient can be only 0 or 1.



(a) An example using decimal numbers

(b) Using binary numbers

```
R = 0;
for i = 0 to n – 1 do
    Left-shift R||A;
    if R ≥ B then
        q_i = 1;
        R = R – B;
    else
        q_i = 0 ;
    end if;
end for;
```

(c) Pseudo-code

Figure 7.28 An algorithm for division.

Given two unsigned n-bit numbers A and B, we wish to design a circuit that produces two n-bit outputs Q and R, where Q is the quotient A/B and R is the remainder. The procedure illustrated in Figure 7.28*b* can be implemented by shifting the digits in A to the left, one digit at a time, into a shift register R. After each shift operation, we compare R with B. If $R \geq B$, a 1 is placed in the appropriate bit position in the quotient and B is subtracted from R. Otherwise, a 0 bit is placed in the quotient. This algorithm is described using pseudo-code in Figure 7.28*c*. The notation $R||A$ is used to represent a $2n$-bit shift register formed using R as the leftmost n bits and A as the rightmost n bits.

In the long-division pseudo-code, each loop iteration results in setting a digit q_i to either 1 or 0. A straightforward way to accomplish this is to shift 1 or 0 into the least-significant bit of Q in each loop iteration. An ASM chart that represents the divider circuit is shown in Figure 7.29. The signal C represents a counter that is initialized to n-1 in the starting state $S1$. In state $S2$, both R and A are shifted to the left, and then in state $S3$, B is subtracted from R if $R \geq B$. The machine changes to state $S4$ when C=0.

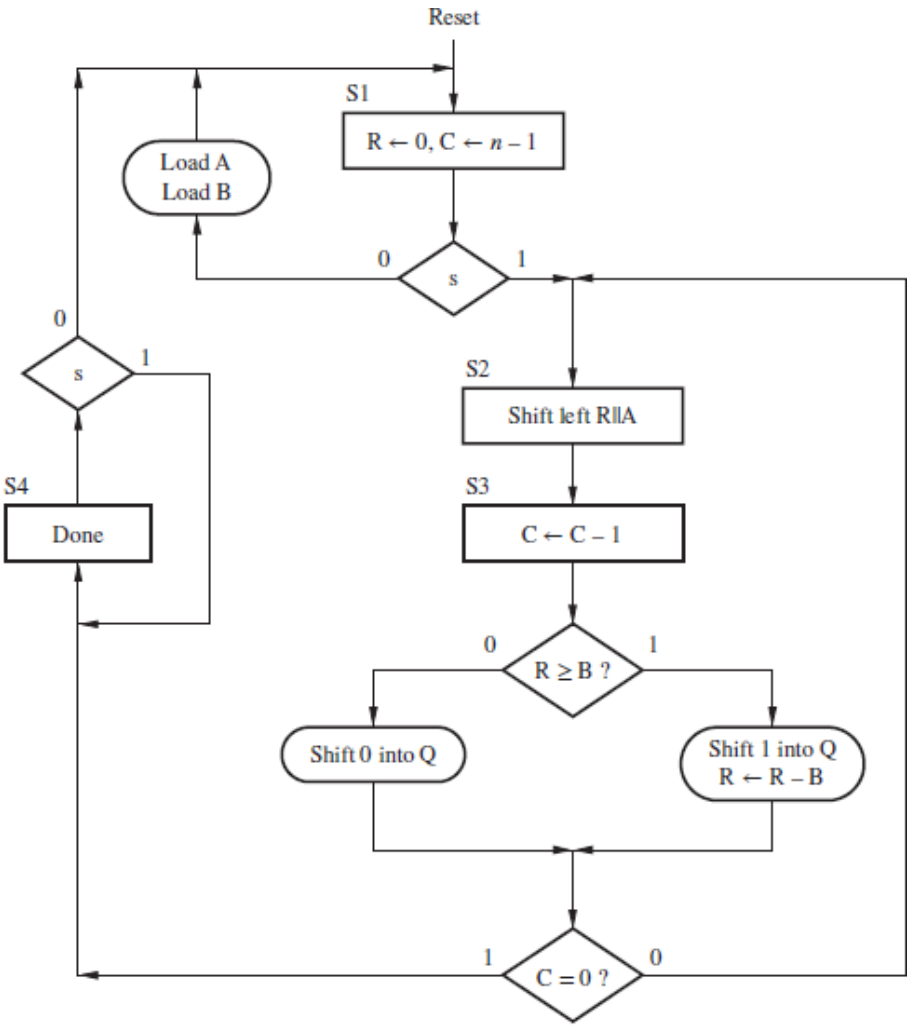


Figure 7.29 ASM chart for the divider.

Datapath Circuit

We need *n*-bit shift registers that shift right to left for *A*, *R*, and Q. An *n*-bit register is needed for *B*, and a subtractor is needed to produce *R*-*B*. We can use an adder module in which the carry-in is set to 1 and *B* is complemented. The carry-out, c_{out}, of this module has the value 1 if the condition $R \geq B$ is true. Hence the carry-out can be connected to the serial input of the shift register that holds Q, so that it is shifted into Q in state *S*3. Since *R* is loaded with 0 in state *S*1 and from the outputs of the adder in state *S*3, a multiplexer is needed for the parallel data inputs on *R*. The datapath circuit is depicted in Figure 7.30. Note that the down-counter needed to implement *C* and the NOR gate that outputs a 1 when *C*=0 are not shown in the figure.

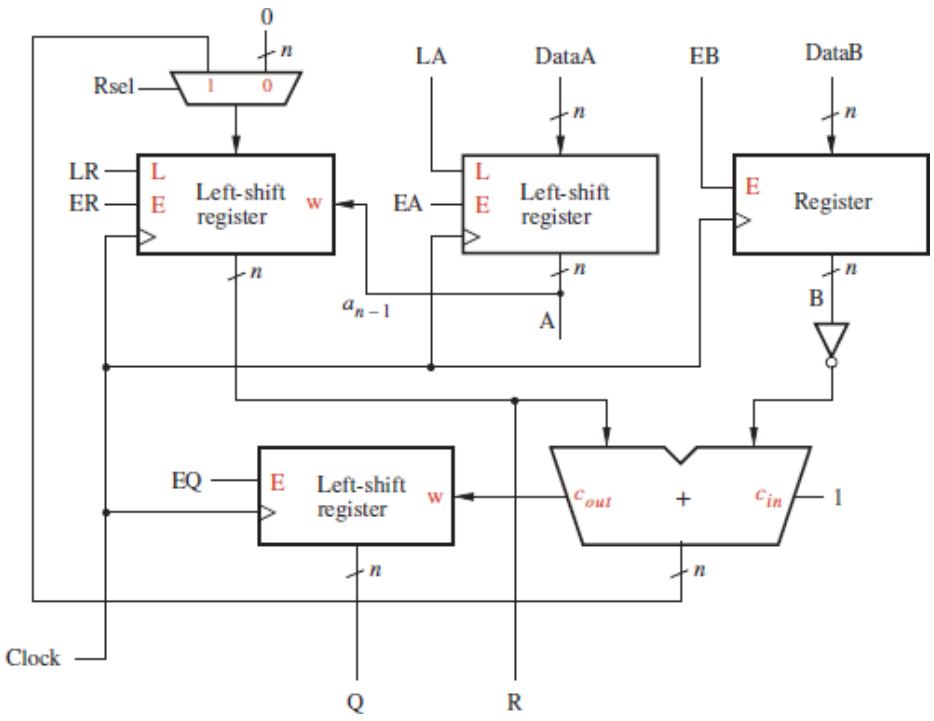


Figure 7.30 Datapath circuit for the divider.

Control Circuit

An ASM chart that shows only the control signals needed for the divider is given in Figure 7.31. In state *S*3 the value of c_{out} determines whether or not the sum output of the adder is loaded into *R*. The shift enable on Q is asserted in state *S*3. We do not have to specify whether 1 or 0 is loaded into Q, because c_{out} is connected to Q's serial input in the datapath circuit. We leave it as an exercise for the reader to write VHDL code that represents the ASM chart in Figure 7.31 and the datapath circuit in Figure 7.30.

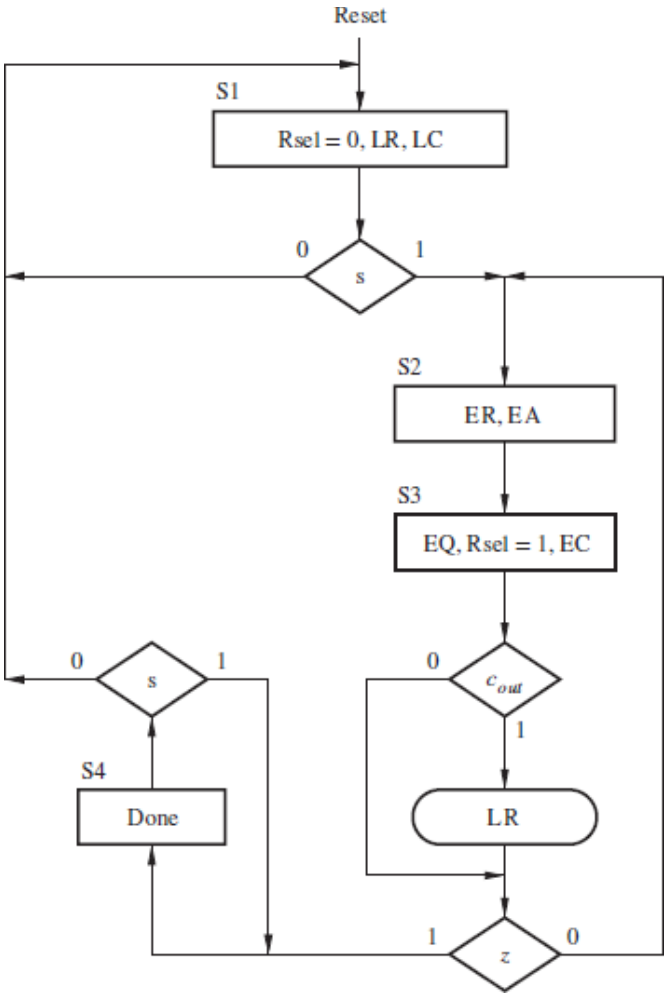


Figure 7.31 ASM chart for the divider control circuit.

Enhancements to the Divider Circuit

Using the ASM chart in Figure 7.31 causes the circuit to loop through states *S*2 and *S*3 for 2*n* clock cycles. If these states can be merged into a single state, then the number of clock cycles needed can be reduced to *n*. In state *S*3, if c_{out}=1, we load the sum output (result of the subtraction) from the adder into *R*, and (assuming *z*=0) change to state *S*2. In state *S*2 we then shift *R* (and *A*) to the left. To combine *S*2 and *S*3 into a new state, called *S*2, we need to be able to place the sum into the leftmost bits of *R* while at the same time shifting the MSB of *A* into the LSB of *R*. This step can be accomplished by using a separate flip-flop for the LSB of *R*. Let the output of this flip-flop be called rr_0. It is initialized to 0 when *s*=0 in state *S*1. Otherwise, the flip-flop is loaded from the MSB of *A*. In state *S*2, if c_{out}=0, *R* is shifted left and rr_0 is shifted into *R*. But if c_{out}=1, *R* is loaded in parallel from the sum outputs of the adder.

Figure 7.32 illustrates how the division example from Figure 7.28*b* can be performed using *n* clock cycles. The table in the figure shows the values of *R*, rr_0, *A*, and Q in each step of the division. In the datapath circuit in Figure 7.30, we use a separate shift register for Q. This register is not actually needed, because the digits in the quotient can be shifted into the least-significant bit of the register used for *A*. In Figure 7.32 the digits of Q that are shifted into *A* are shown in red. The first row in the table represents loading of initial data into registers *A* (and *B*) and clearing *R* and rr_0 to 0. In the second row of the table, labeled clock cycle 0, the diagonal red arrow shows that the leftmost bit of *A* (1) is shifted into rr_0. The number in $R||rr_0$ is now 000000001, which is smaller than *B* (1001). In clock cycle 1, rr_0 is shifted into *R*, and the MSB of *A* is shifted into rr_0. Also, as shown in red, a 0 is shifted into the LSB of Q (*A*). The number in $R||rr_0$ is now 000000010, which is still smaller than *B*. Hence, in clock cycle 2 the same actions are performed as for clock cycle 1. These actions are also performed in clock cycles 3 and 4, at which point $R||rr_0$ = 000010001. Since this is larger than *B*, in clock cycle 5 the result of the subtraction 000010001 - 1001 = 00001000 is loaded into *R*. The MSB of *A* (1) is still shifted into rr_0, and a 1 is shifted into Q. In clock cycles 6, 7, and 8, the number in $R||rr_0$ is larger than *B*; hence in each of these cycles the result of the subtraction $R||rr_0$ - *B* is loaded into *R*, and a 1 is loaded into Q. After clock cycle 8 the correct result, *Q* = 00001111 and *R*=00000101, is obtained. The bit rr_0 is not a part of the final result.

	Clock cycle	R	rr_0	A/Q
	Load A, B	0 0 0 0 0 0 0 0	0	1 0 0 0 1 1 0 0
0	Shift left	0 0 0 0 0 0 0 0	1	0 0 0 1 1 0 0 0
1	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 0 0 0 1	0	0 0 1 1 0 0 0 0
2	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 0 0 1 0	0	0 1 1 0 0 0 0 0
3	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 0 1 0 0	0	1 1 0 0 0 0 0 0
4	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 1 0 0 0	1	1 0 0 0 0 0 0 0
5	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 1 0 0 0	1	0 0 0 0 0 0 0 1
6	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 1 0 0 0	0	0 0 0 0 0 0 1 1
7	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 0 1 1 1	0	0 0 0 0 0 1 1 1
8	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 0 1 0 1	0	0 0 0 0 1 1 1 1

Figure 7.32 An example of division using *n* = 8 clock cycles.

An ASM chart that shows the values of the required control signals for the enhanced divider is depicted in Figure 7.33. The signal ER0 is used in conjunction with the flip-flop that has the output rr_0. When ER0 = 0, the value 0

is loaded into the flip-flop. When ER0 is set to 1, the MSB of shift register *A* is loaded into the flip-flop. In state *S*1, if *s*=0, then *LR* is asserted to initialize *R* to 0. Registers *A* and *B* can be loaded with data from external inputs. When *s* changes to 1, the machine makes a transition to state *S*2 and at the same time shifts *R*||*R*0||A to the left. In state *S*2, if c_{out} = 1, then *R* is loaded in parallel from the sum outputs of the adder. At the same time, *R*0||*A* is shifted left (rr_0 is not shifted into *R* in this case). If c_{out}=0, then *R*||*R*0||*A* is shifted left. The ASM chart shows how the parallel-load and enable inputs on the registers have to be controlled to achieve the desired operation.

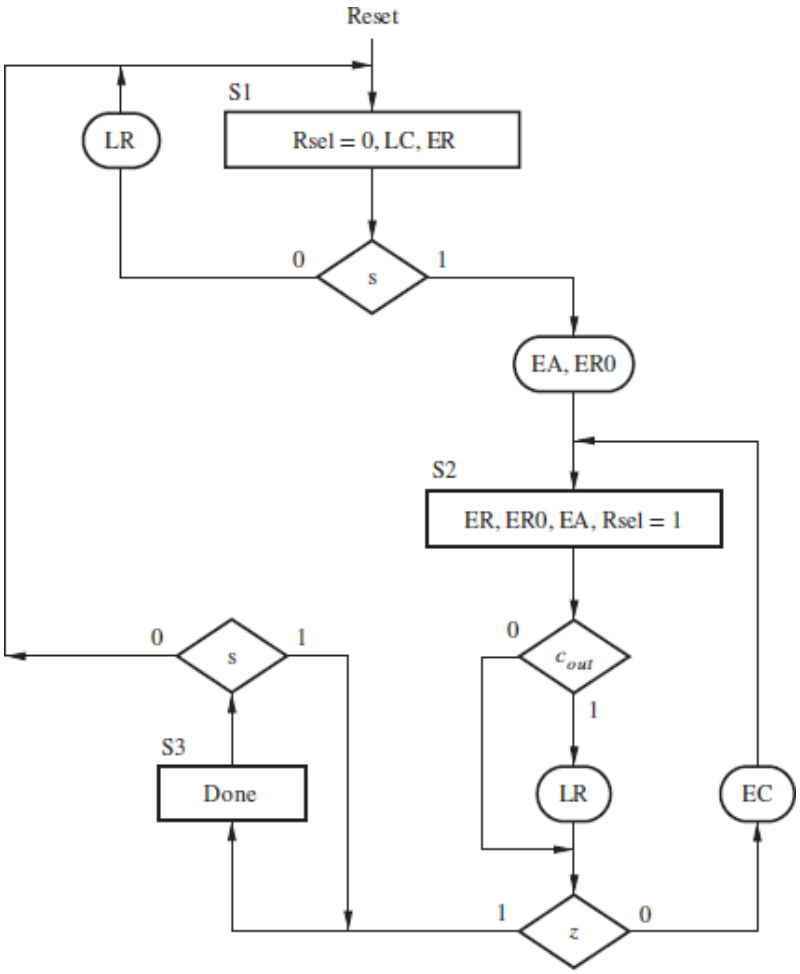


Figure 7.33 ASM chart for the enhanced divider control circuit.

The datapath circuit for the enhanced divider is illustrated in Figure 7.34. As discussed for Figure 7.32, the digits of the quotient Q are shifted into register *A*. Note that one of the *n*-bit data inputs on the adder module is composed of the *n*-1 least-significant bits in register *R* concatenated with bit rr_0 on the right.

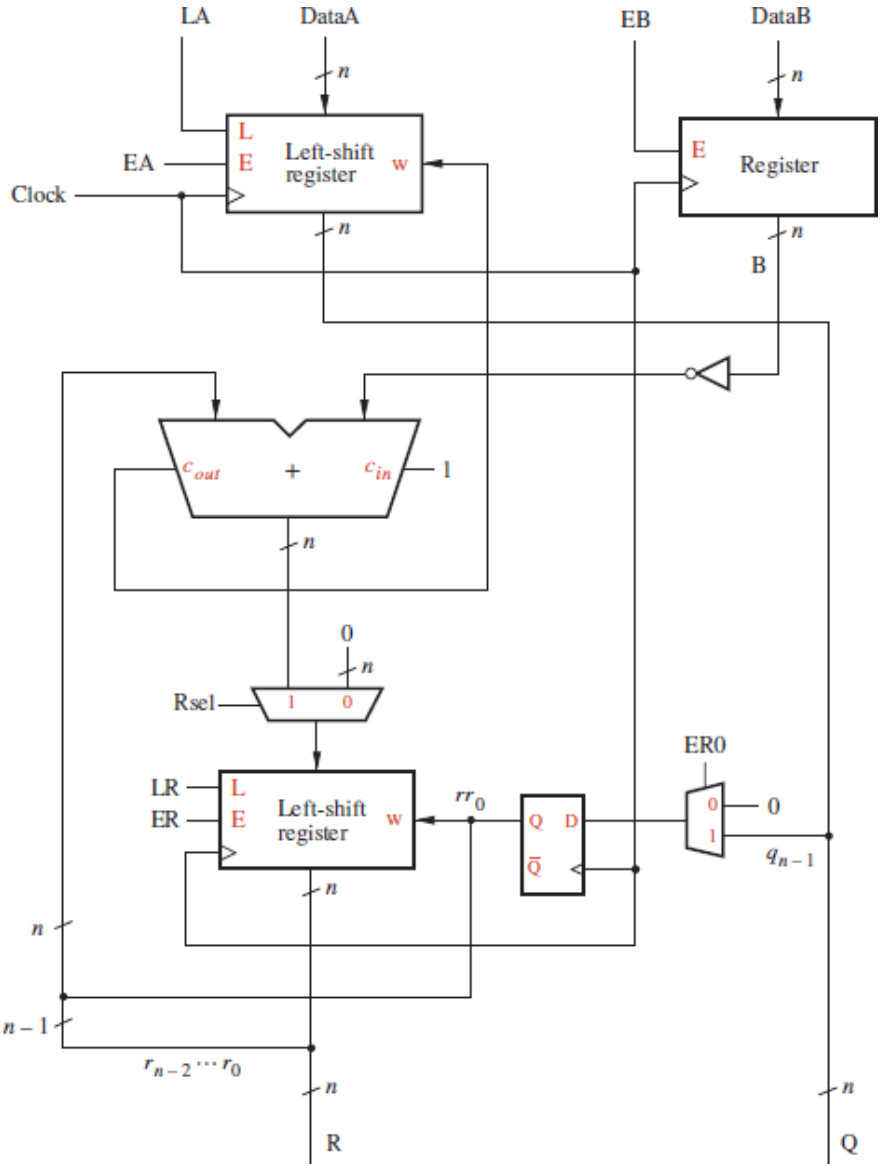


Figure 7.34 Datapath circuit for the enhanced divider.

VHDL Code

Figure 7.35 shows VHDL code that represents the enhanced divider. The generic parameter *N* sets the number of bits in the operands. The *FSM_transitions* and *FSM_outputs* processes describe the control circuit, as in the previous examples. The shift registers and counters in the datapath circuit are instantiated at the bottom of the code. The signal rr_0 in Figure 7.32 is represented in the code by the signal *R*0. This signal is implemented as the output of the *muxdff* component; the code for this subcircuit is shown in Figure 5.43. Note that the adder that produces the *Sum* signal has one input defined as the concatenation of

R with *R*0. The multiplexer needed for the input to *R* is represented by the *DataR* signal. Instead of describing this multiplexer using a FOR GENERATE statement as in the previous examples, we have used the conditional signal assignment shown at the end of the code.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all ;
USE work.components.all ;

ENTITY divider IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT( Clock      : IN      STD_LOGIC ;
          Resetn     : IN      STD_LOGIC ;
          s, LA, EB  : IN      STD_LOGIC ;
          DataA      : IN      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
          DataB      : IN      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
          R, Q       : BUFFER  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
          Done       : OUT     STD_LOGIC ) ;
END divider ;

ARCHITECTURE Behavior OF divider IS
    TYPE State_type IS ( S1, S2, S3 ) ;
    SIGNAL y : State_type ;
    SIGNAL Zero, Cout, z : STD_LOGIC ;
    SIGNAL EA, Rsel, LR, ER, ER0, LC, EC, R0 : STD_LOGIC ;
    SIGNAL A, B, DataR : STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
    SIGNAL Sum : STD_LOGIC_VECTOR(N DOWNTO 0) ; -- adder outputs
    SIGNAL Count : INTEGER RANGE 0 TO N-1 ;
BEGIN
    FSM_transitions: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN y <= S1 ;
        ELSIF (Clock'EVENT AND Clock = '1' ) THEN
            CASE y IS
                WHEN S1 =>
                    IF s = '0' THEN y <= S1 ; ELSE y <= S2 ; END IF ;
                WHEN S2 =>
                    IF z = '0' THEN y <= S2 ; ELSE y <= S3 ; END IF ;
                WHEN S3 =>
                    IF s = '1' THEN y <= S3 ; ELSE y <= S1 ; END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

... continued in Part b
```

Figure 7.35 VHDL code for the divider circuit (Part *a*).

```
    FSM_outputs: PROCESS ( s, y, Cout, z )
    BEGIN
        LR <= '0' ; ER <= '0' ; ER0 <= '0' ;
        LC <= '0' ; EC <= '0' ; EA <= '0' ; Done <= '0' ;
        Rsel <= '0' ;
        CASE y IS
            WHEN S1 =>
                LC <= '1' ; EC <= '1' ; ER <= '1' ;
                IF s = '0' THEN
                    LR <= '1' ;
                ELSE
                    EA <= '1' ; ER0 <= '1' ;
                END IF ;
            WHEN S2 =>
                Rsel <= '1' ; ER <= '1' ; ER0 <= '1' ; EA <= '1' ;
                IF Cout = '1' THEN LR <= '1' ; ELSE LR <= '0' ; END IF ;
                IF z = '0' THEN EC <= '1' ; ELSE EC <= '0' ; END IF ;
            WHEN S3 =>
                Done <= '1' ;
        END CASE ;
    END PROCESS ;
    -- define the datapath circuit
    Zero <= '0' ;
    RegB: regne GENERIC MAP ( N => N )
        PORT MAP ( DataB, Resetn, EB, Clock, B ) ;
    ShiftR: shiftlne GENERIC MAP ( N => N )
        PORT MAP ( DataR, LR, ER, R0, Clock, R ) ;
    FF_R0: muxdff PORT MAP ( Zero, A(N–1), ER0, Clock, R0 ) ;
    ShiftA: shiftlne GENERIC MAP ( N => N )
        PORT MAP ( DataA, LA, EA, Cout, Clock, A ) ;
    Q <= A ;
    Counter: downcnt GENERIC MAP ( modulus => N )
        PORT MAP ( Clock, EC, LC, Count ) ;
    z <= '1' WHEN Count = 0 ELSE '0' ;

    Sum <= R & R0 + (NOT B +1) ;
    Cout <= Sum(N) ;
    DataR <= (OTHERS => '0' ) WHEN Rsel = '0' ELSE Sum ;
END Behavior ;
```

Figure 7.35b VHDL code for the divider circuit (Part *b*).

A simulation result for the circuit produced from the code is given in Figure 7.36. The data $A = A6$ and $B=8$ is loaded, and then s is set to 1. The circuit changes to state $S2$ and concurrently shifts R, $R0$, and A to the left. The output of the shift register that holds A is labeled Q in the simulation results because this shift register contains the quotient when the division operation is complete. On the first three active clock edges in state $S2$, the number represented by $R||R0$ is less than the number in B (8); hence $R||R0||A$ is shifted left on each clock edge, and 0 is shifted into Q. In the fourth consecutive clock cycle for which the FSM has been in state $S2$, the contents of R are 00000101 $=(5)_{10}$, and $R0$ is 0; hence

R||R0 = 000001010 = $(10)_{10}$. On the next active clock edge, the output of the adder, which is 10 - 8 =2, is loaded into R, and 1 is shifted into Q. After n clock cycles in state $S2$, the circuit changes to state $S3$, and the correct result, Q = 14 = $(20)_{10}$ and R=6, is obtained.

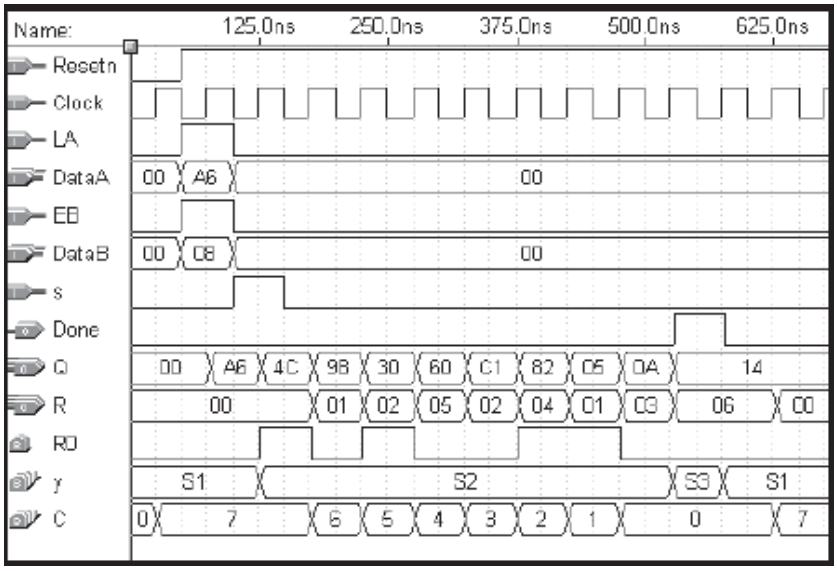


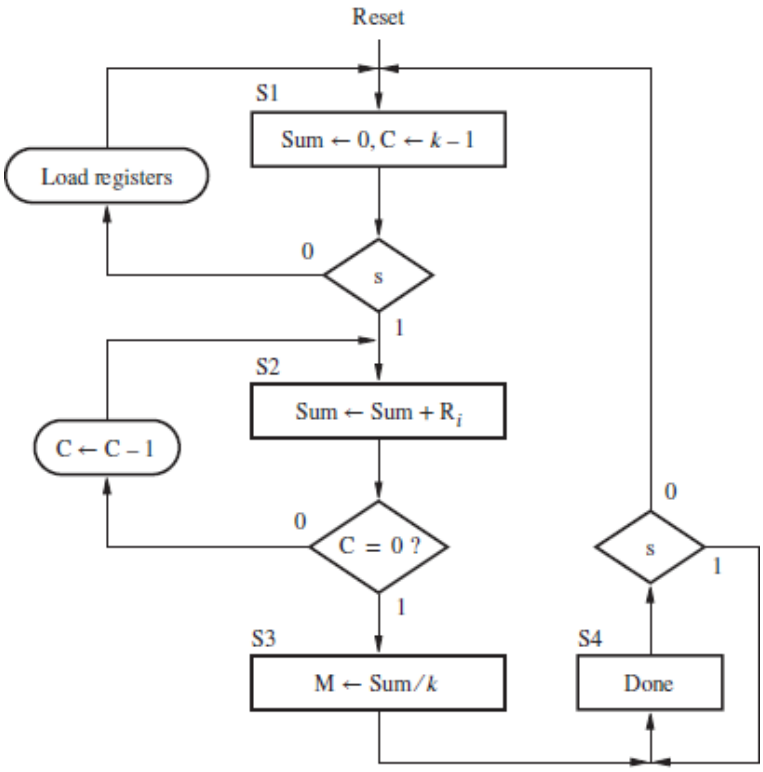
Figure 7.36 Simulation results for the divider circuit.

7.6 Arithmetic Mean

Assume that k n-bit numbers are stored in a set of registers R_0, …, R_{k-1}. We wish to design a circuit that computes the mean M of the numbers in the registers. The pseudo-code for a suitable algorithm is shown in Figure 7.37*a*. Each iteration of the loop adds the contents of one of the registers, denoted R_i, to a *Sum* variable. After the sum is computed, M is obtained as *Sum*/k. We assume that integer division is used, so a remainder R, not shown in the code, is produced as well.

```
Sum = 0;
for i = k – 1 down to 0 do
    Sum = Sum + Ri
end for;
M = Sum ÷ k;
```

(a) Pseudo-code



(b) ASM chart

Figure 7.37 An algorithm for finding the mean of *k* numbers.

An ASM chart is given in Figure 7.37*b*. While the start input, *s*, is 0, the registers can be loaded from external inputs. When *s* becomes 1, the machine changes to state *S*2, where it remains while $C \neq 0$, and computes the summation (*C* is a counter that represents *i* in Figure 7.37*a*). When *C*=0, the machine changes to state *S*3 and computes *M* = *Sum*/*k*. From the previous example, we know that the division operation requires multiple clock cycles, but we have chosen not to indicate this in the ASM chart. After computing the division operation, state *S*4 is entered and *Done* is set to 1.

Datapath Circuit

The datapath circuit for this task is more complex than in our previous examples. It is depicted in Figure 7.38. We need a register with an enable input to hold *Sum*. For simplicity, assume that the sum can be represented in *n* bits without overflowing. A multiplexer is required on the data inputs on the *Sum* register, to select 0 in state *S*1 and the sum outputs of an adder in state *S*2. The *Sum* register provides one of the data inputs to the adder. The other input has to be selected from the data outputs of one of the *k* registers. One way to select among the registers is to connect them to the data inputs of a *k*-to-1 multiplexer that is connected to the adder. The select lines on the multiplexer can be controlled by the counter *C*. To compute the division operation, we can use the divider circuit designed in the previous section.

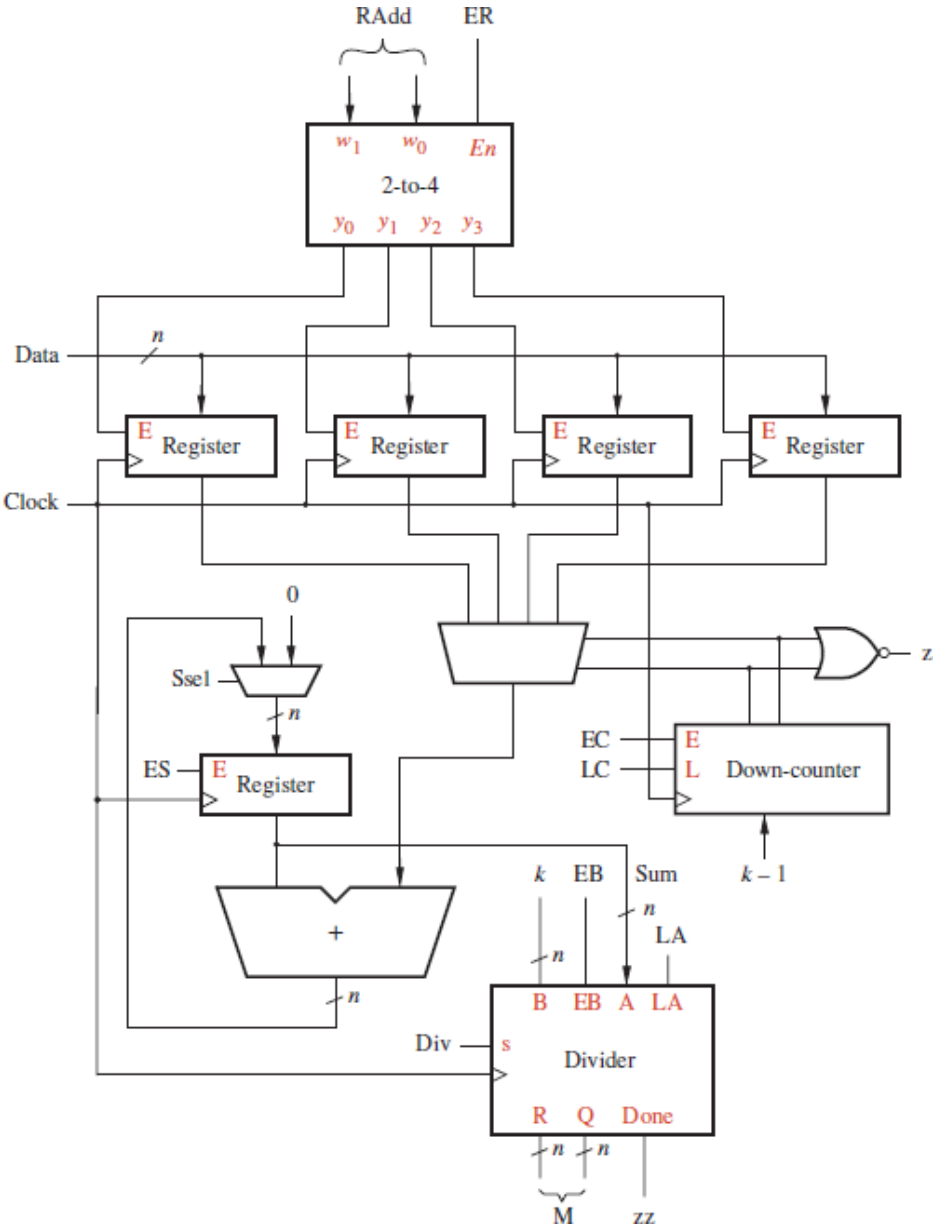


Figure 7.38 Datapath circuit for the mean operation.

The circuit in Figure 7.38 is based on k=4, but the same circuit structure can be used for larger values of k. Note that the enable inputs on the registers R_0 through R_3 are connected to the outputs of a 2-to-4 decoder that has the two-bit input *RAdd*, which stands for "register address." The decoder enable input is driven by the *ER* signal. All registers are loaded from the same input lines, *Data*. Since k=4, we could perform the division operation simply by shifting *Sum* two bits to the right, which can be done in one clock cycle with a shift register that shifts by two digits. To obtain a more general circuit that works for any value of k, we use the divider circuit designed in Section 7.5.

Control Circuit

Figure 7.39 gives an ASM chart for the FSM needed to control the circuit in Figure 7.38. While in state *S*1, data can be loaded into registers $R_0,\ldots,R_{k-1}$. But no control signals have to be asserted for this purpose, because the registers are loaded under control of the *ER* and *RAdd* inputs, as discussed above. When *s*=1, the FSM changes to state *S*2, where it asserts the enable *ES* on the *Sum* register and allows *C* to decrement. When the counter reaches 0 (*z*=1), the machine enters state *S*3, where it asserts the *LA* and *EB* signals to load the *Sum* and *k* into the *A* and *B* inputs of the divider circuit, respectively. The FSM then enters state *S*4 and asserts the *Div* signal to start the division operation. When it is finished, the divider circuit sets *zz* =1, and the FSM moves to state *S*5. The mean *M* appears on the Q and *R* outputs of the divider circuit. The *Div* signal must still be asserted in state *S*5 to prevent the divider circuit from reinitializing its registers. Note that in the ASM chart in Figure 7.37*b*, only one state is shown for computing *M* = *Sum*/*k*, but in Figure 7.39, states *S*3 and *S*4 are used for this purpose. It is possible to combine states *S*3 and *S*4, which we will leave as an exercise for the reader (Problem 7.10).

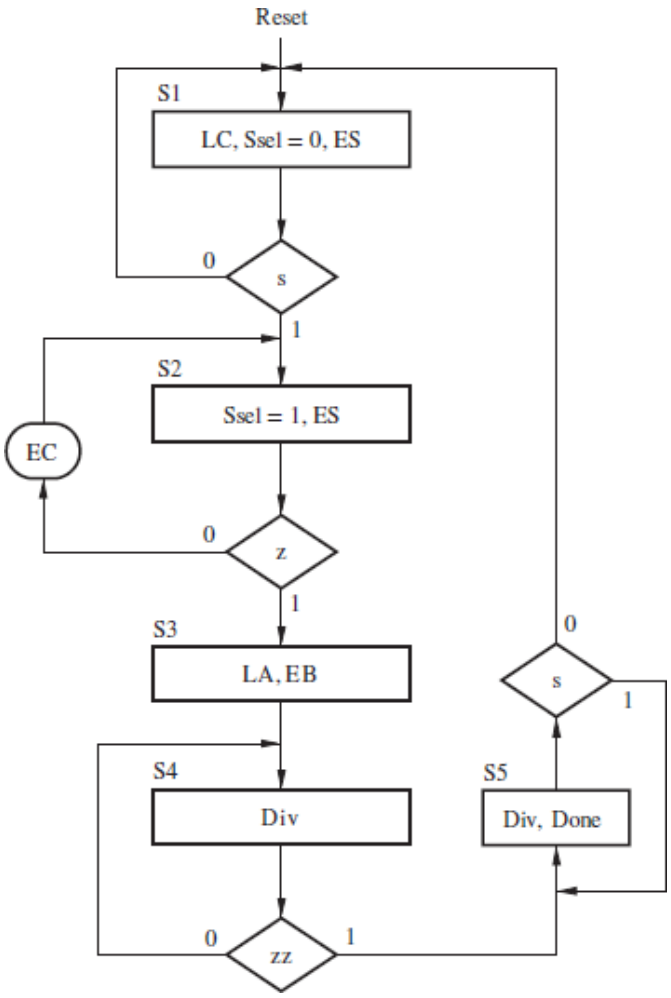


Figure 7.39 ASM chart for the mean operation control circuit.

7.7 Sort Operation

Given a list of k unsigned n-bit numbers stored in a set of registers R_0, …, R_{k-1}, we wish to design a circuit that can sort the list (contents of the registers) in ascending order. Pseudo-code for a simple sorting algorithm is shown in Figure 7.40. It is based on finding the smallest number in the sublist R_i, …, R_{k-1} and moving that number into R_i, for i=1, 2, …, k-2. Each iteration of the outer loop places the number in R_i into A. Each iteration of the inner loop compares this number to the contents of another register R_j. If the number in R_j is smaller than A, the contents of R_i and R_j are swapped and A is changed to hold the new contents of R_i.

```
for i = 0 to k – 2 do
    A = R_i;
    for j = i C 1 to k – 1 do
        B = R_j;
        if B < A then
            R_i = B;
            R_j = A;
            A = R_i;
        end if;
    end for;
end for;
```

Figure 7.40 Pseudo-code for the sort operation.

An ASM chart that represents the sorting algorithm is shown in Figure 7.41. In the initial state *S*1, while *s*=0 the registers are loaded from external data inputs and a counter C_i that represents *i* in the outer loop is cleared. When the machine changes to state *S*2, *A* is loaded with the contents of R_i. Also, C_j, which represents *j* in the inner loop, is initialized to the value of *i*. State *S*3 is used to initialize *j* to the value *i*+1, and state *S*4 loads the value of R_j into *B*. In state *S*5, *A* and *B* are compared, and if *B*<*A*, the machine moves to state *S*6. States *S*6 and *S*7 swap the values of R_i and R_j. State *S*8 loads *A* from R_i. Although this step is necessary only for the case where *B*<*A*, the flow of control is simpler if this operation is performed in both cases. If C_j is not equal to *k*-1, the machine changes from *S*8 to *S*4, thus remaining in the inner loop. If C_j = *k*-1 and C_i is not equal to *k*-2, then the machine stays in the outer loop by changing to state *S*2.

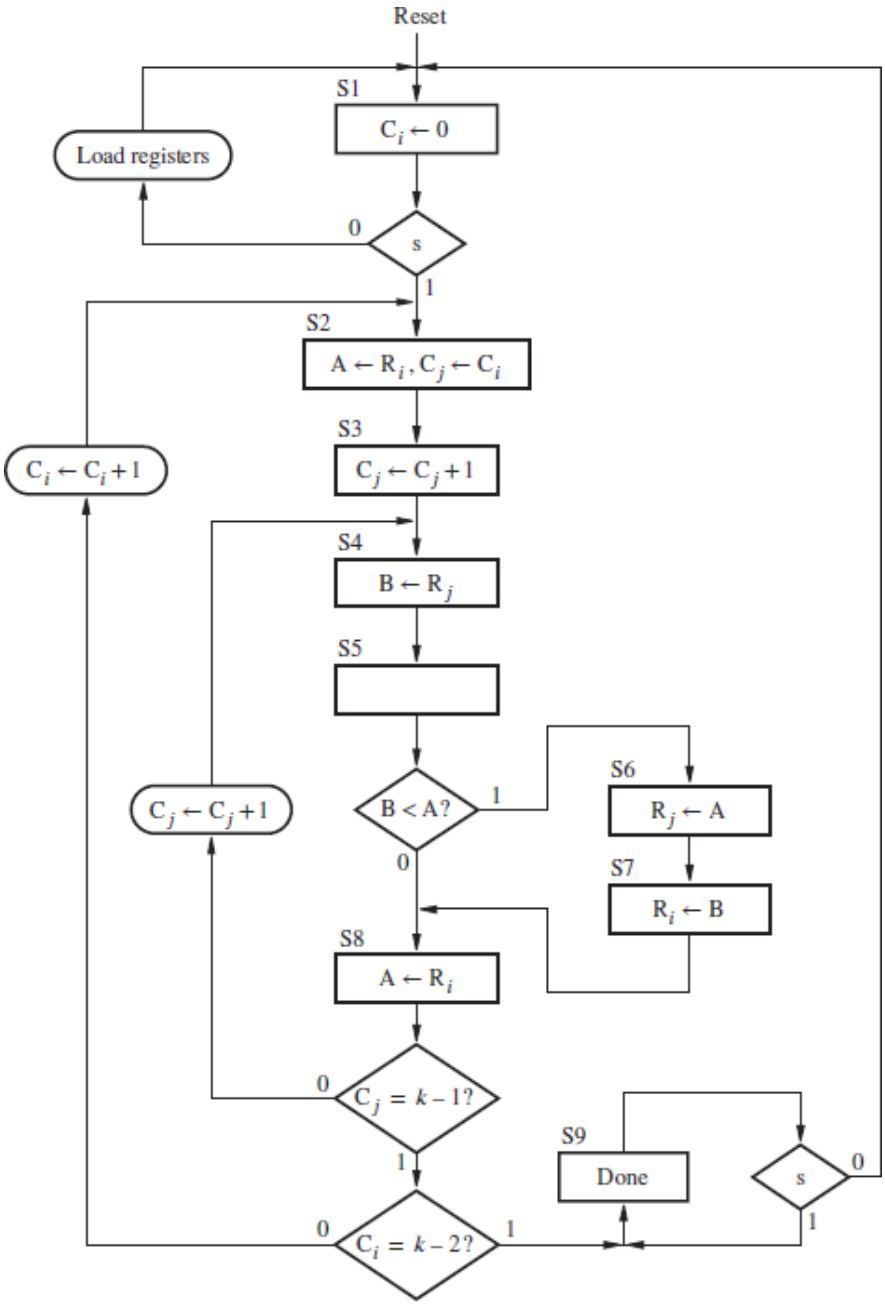


Figure 7.41 ASM chart for the sort operation.

Datapath Circuit

There are many ways to implement a datapath circuit that meets the requirements of the ASM chart in Figure 7.41. One possibility is illustrated in Figures 7.42 and 7.43. Figure 7.42 shows how the registers R_0, …, R_{k-1} can be connected to registers A and B using 4-to-1 multiplexers. We assume the value k=4 for simplicity. Registers A and B are connected to a comparator subcircuit

and, through multiplexers, back to the inputs of the registers R_0, …, R_{k-1}. The registers can be loaded with initial (unsorted) data using the *DataIn* lines. The data is written (loaded) into each register by asserting the *WrInit* control signal and placing the address of the register on the *RAdd* input. The tri-state driver driven by the *Rd* control signal is used to output the contents of the registers on the *DataOut* output.

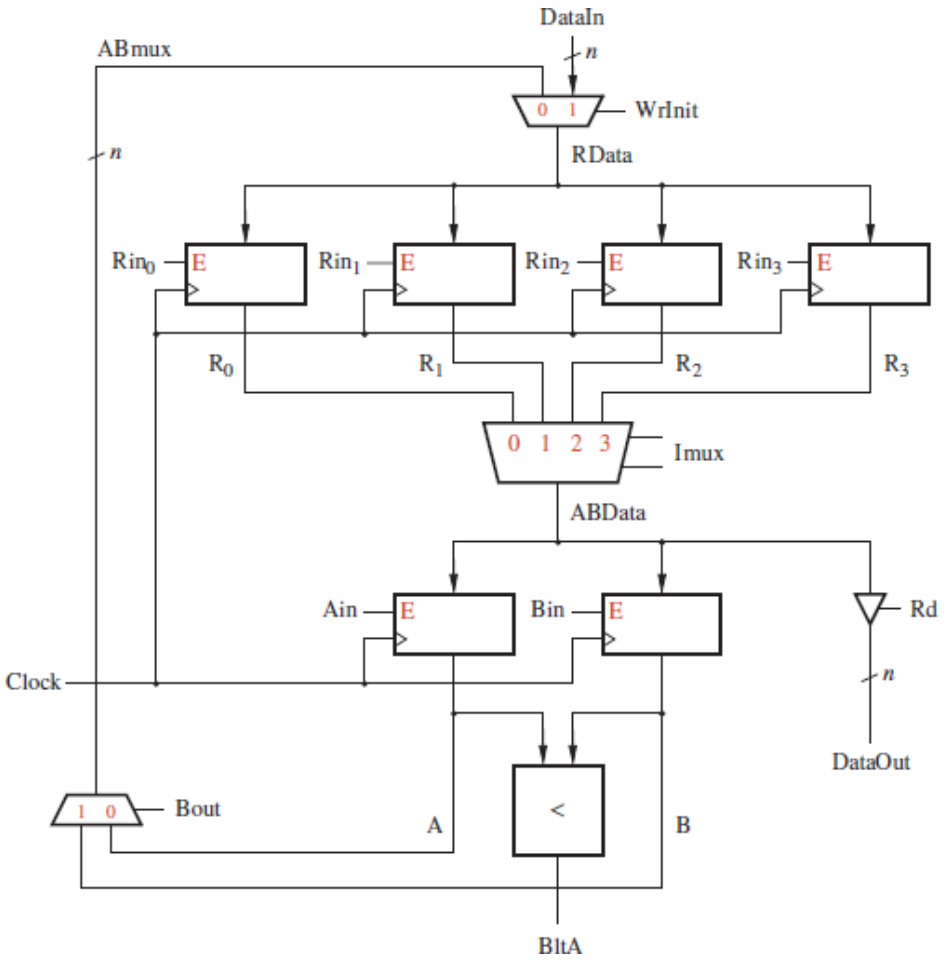


Figure 7.42 A part of the datapath circuit for the sort operation.

The signals Rin_0, …, Rin_{k-1} are controlled by the 2-to-4 decoder shown in Figure 7.43. If *Int* =1, the decoder is driven by one of the counters C_i or C_j. If *Int* =0, then the decoder is driven by the external input *RAdd*. The signals z_i and z_j are set to 1 if C_i=*k*-2 and C_j = *k*-1, respectively. An ASM chart that shows the control signals used in the datapath circuit is given in Figure 7.44.

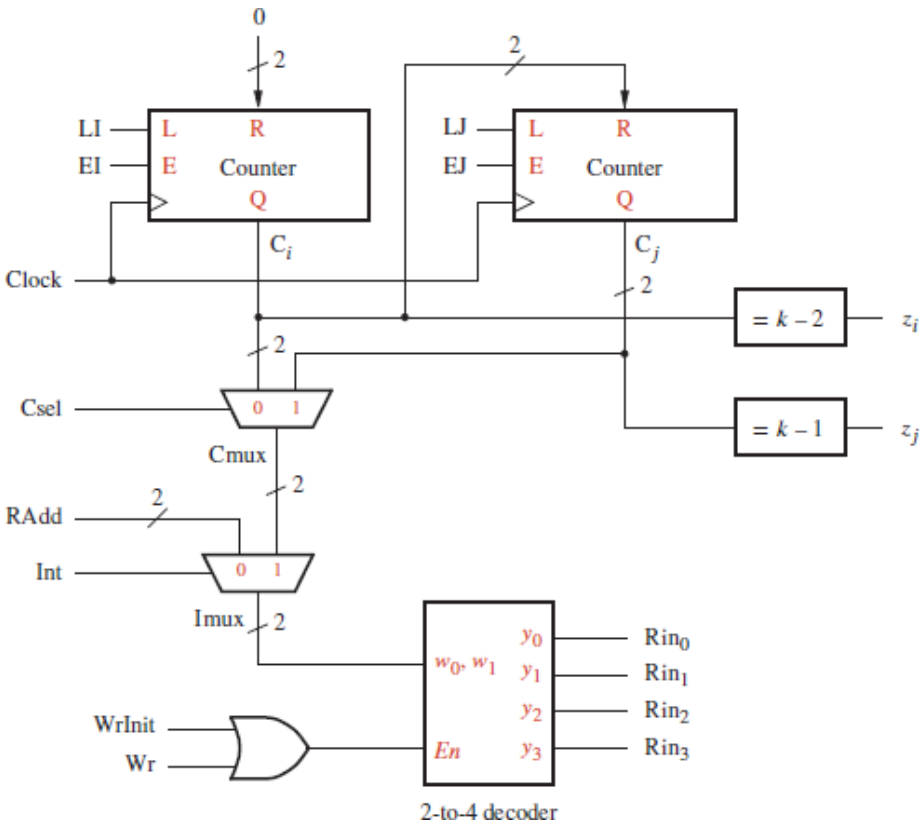


Figure 7.43 A part of the datapath circuit for the sort operation.

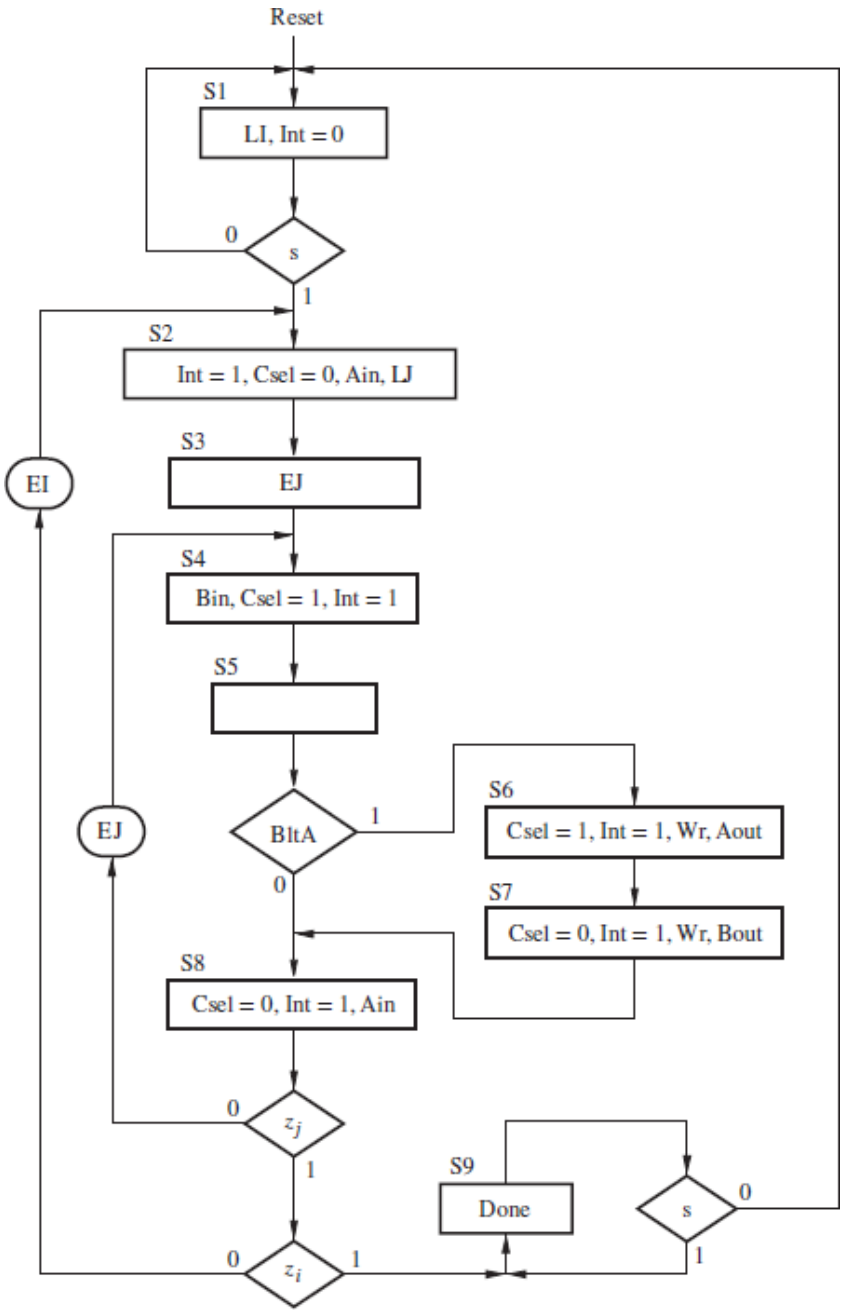


Figure 7.44 ASM chart for the control circuit.

VHDL Code

VHDL code for the sorting operation is presented in Figure 7.45. Instead of defining separate signals called R_0, …, R_3 for the register outputs, we have chosen to specify the registers as an array. This approach allows the registers to be referred to as $R(i)$ in a FOR GENERATE statement that instantiates each

register. The array of registers is defined in two steps. First, a user-defined type, for which we have chosen the name *RegArray*, is defined in the statement

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.components.all ;

ENTITY sort IS
    GENERIC ( N : INTEGER := 4 ) ;
    PORT ( Clock, Resetn   : IN      STD_LOGIC ;
           s, WrInit, Rd   : IN      STD_LOGIC ;
           DataIn          : IN      STD_LOGIC_VECTOR(N–1 DOWNTO 0) ;
           RAdd            : IN      INTEGER RANGE 0 TO 3 ;
           DataOut         : BUFFER  STD_LOGIC_VECTOR(N–1 DOWNTO 0) ;
           Done            : BUFFER  STD_LOGIC ) ;
END sort ;

ARCHITECTURE Behavior OF sort IS
    TYPE State_type IS ( S1, S2, S3, S4, S5, S6, S7, S8, S9 ) ;
    SIGNAL y : State_type ;
    SIGNAL Ci, Cj : INTEGER RANGE 0 TO 3 ;
    SIGNAL Rin : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
    TYPE RegArray IS
        ARRAY(3 DOWNTO 0) OF STD_LOGIC_VECTOR(N–1 DOWNTO 0) ;
    SIGNAL R : RegArray ;
    SIGNAL RData, ABMux : STD_LOGIC_VECTOR(N–1 DOWNTO 0) ;
    SIGNAL Int, Csel, Wr, BltA : STD_LOGIC ;
    SIGNAL CMux, IMux : INTEGER RANGE 0 TO 3 ;
    SIGNAL Ain, Bin, Aout, Bout : STD_LOGIC ;
    SIGNAL LI, LJ, EI, EJ, zi, zj : STD_LOGIC ;
    SIGNAL Zero : INTEGER RANGE 3 DOWNTO 0 ; - - parallel data for Ci = 0
    SIGNAL A, B, ABData : STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
BEGIN
    FSM_transitions: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= S1 ;
        ELSIF (Clock'EVENT AND Clock = '1' ) THEN
            CASE y IS
                WHEN S1 => IF S = '0' THEN y <= S1 ;
                    ELSE y <= S2 ; END IF ;
                WHEN S2 => y <= S3 ;
                WHEN S3 => y <= S4 ;
                WHEN S4 => y <= S5 ;
```

… continued in Part *b*

Figure 7.45a VHDL code for the sorting circuit (Part *a*).

```
                    WHEN S5 => IF BltA = '1' THEN y <= S6 ;
                        ELSE y <= S8 ; END IF ;
                    WHEN S6 => y <= S7 ;
                    WHEN S7 => y <= S8 ;
                    WHEN S8 =>
                        IF zj = '0' THEN y <= S4 ;
                        ELSIF zi = '0' THEN y <= S2 ;
                        ELSE y <= S9 ;
                        END IF ;
                    WHEN S9 => IF s = '1' THEN y <= S9 ; ELSE y <= S1 ; END IF ;
                END CASE ;
            END IF ;
        END PROCESS ;
        -- define the outputs generated by the FSM
        Int <= '0' WHEN y = S1 ELSE '1' ;
        Done <= '1' WHEN y = S9 ELSE '0' ;
        FSM_outputs: PROCESS ( y, zi, zj )
        BEGIN
            LI <= '0' ; LJ <= '0' ; EI <= '0' ; EJ <= '0' ; Csel <= '0' ;
            Wr <= '0' ; Ain <= '0' ; Bin <= '0' ; Aout <= '0' ; Bout <= '0' ;
            CASE y IS
                WHEN S1 => LI <= '1' ; EI <= '1' ;
                WHEN S2 => Ain <= '1' ; LJ <= '1' ; EJ <= '1' ;
                WHEN S3 => EJ <= '1' ;
                WHEN S4 => Bin <= '1' ; Csel <= '1' ;
                WHEN S5 => -- no outputs asserted in this state
                WHEN S6 => Csel <= '1' ; Wr <= '1' ; Aout <= '1' ;
                WHEN S7 => Wr <= '1' ; Bout <= '1' ;
                WHEN S8 => Ain <= '1' ;
                    IF zj = '0' THEN
                        EJ <= '1' ;
                    ELSE
                        EJ <= '0' ;
                        IF zi = '0' THEN
                            EI <= '1' ;
                        ELSE
                            EI <= '0' ;
                        END IF;
                    END IF ;
                WHEN S9 => -- Done is assigned 1 by conditional signal assignment
            END CASE ;
        END PROCESS ;
```

… continued in Part *c*

Figure 7.45b VHDL code for the sorting circuit (Part *b*).

```
    -- define the datapath circuit
    Zero <= 0 ;
    GenReg: FOR i IN 0 TO 3 GENERATE
        Reg: regne GENERIC MAP ( N => N )
            PORT MAP ( RData, Resetn, Rin(i), Clock, R(i) ) ;
    END GENERATE ;
    RegA: regne GENERIC MAP ( N => N )
        PORT MAP ( ABData, Resetn, Ain, Clock, A ) ;
    RegB: regne GENERIC MAP ( N => N )
        PORT MAP ( ABData, Resetn, Bin, Clock, B ) ;
    BltA <= '1' WHEN B < A ELSE '0' ;
    ABMux <= A WHEN Bout = '0' ELSE B ;
    RData <= ABMux WHEN WrInit = '0' ELSE DataIn ;
    OuterLoop: upcount GENERIC MAP ( modulus => 4 )
        PORT MAP ( Resetn, Clock, EI, LI, Zero, Ci ) ;
    InnerLoop: upcount GENERIC MAP ( modulus => 4 )
        PORT MAP ( Resetn, Clock, EJ, LJ, Ci, Cj ) ;
    CMux <= Ci WHEN Csel = '0' ELSE Cj ;
    IMux <= Cmux WHEN Int = '1' ELSE Radd ;
    WITH IMux Select
        ABData <= R(0) WHEN 0,
                  R(1) WHEN 1,
                  R(2) WHEN 2,
                  R(3) WHEN OTHERS ;
    RinDec: PROCESS ( WrInit, Wr, IMux )
    BEGIN
        IF (WrInit OR Wr) = '1' THEN
            CASE IMux IS
                WHEN 0 => Rin <= "0001" ;
                WHEN 1 => Rin <= "0010" ;
                WHEN 2 => Rin <= "0100" ;
                WHEN OTHERS => Rin <= "1000" ;
            END CASE ;
        ELSE Rin <= "0000" ;
        END IF ;
    END PROCESS ;
    Zi <= '1' WHEN Ci = 2 ELSE '0' ;
    Zj <= '1' WHEN Cj = 3 ELSE '0' ;
    DataOut <= (OTHERS => 'Z') WHEN Rd = '0' ELSE ABData ;
END Behavior ;
```

Figure 7.45c VHDL code for the sorting circuit (Part *c*).

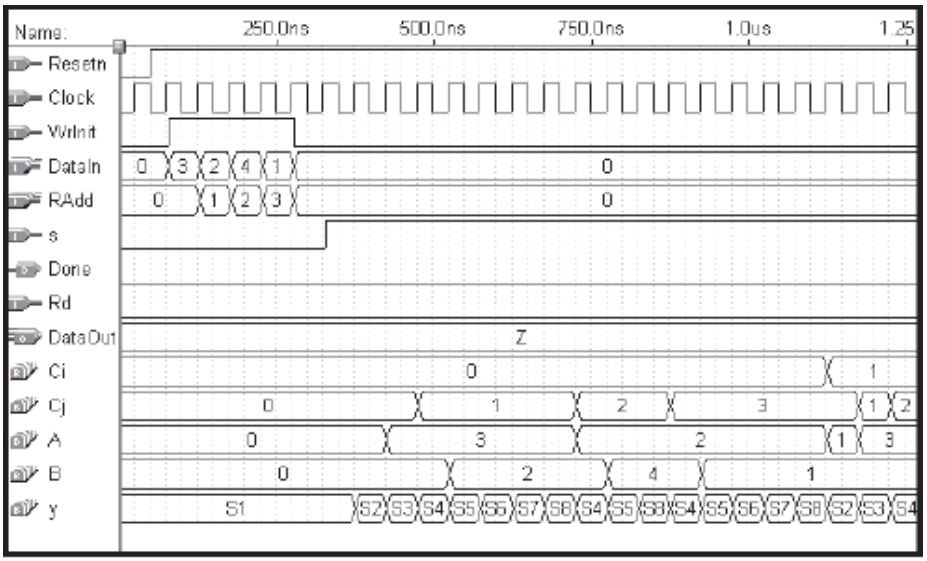
TYPE *RegArray* IS ARRAY(3 DOWNTO 0) OF STD_LOGICVECTOR(N-1 DOWNTO 0)

This statement specifies that the type *RegArray* represents an array of four STD_LOGIC_ VECTOR signals. The STD_LOGIC_VECTOR type is also defined as an array in the IEEE standard; it is an array of STD_LOGIC signals. The *R* signal is defined as an array with four elements of the *RegArray* type.

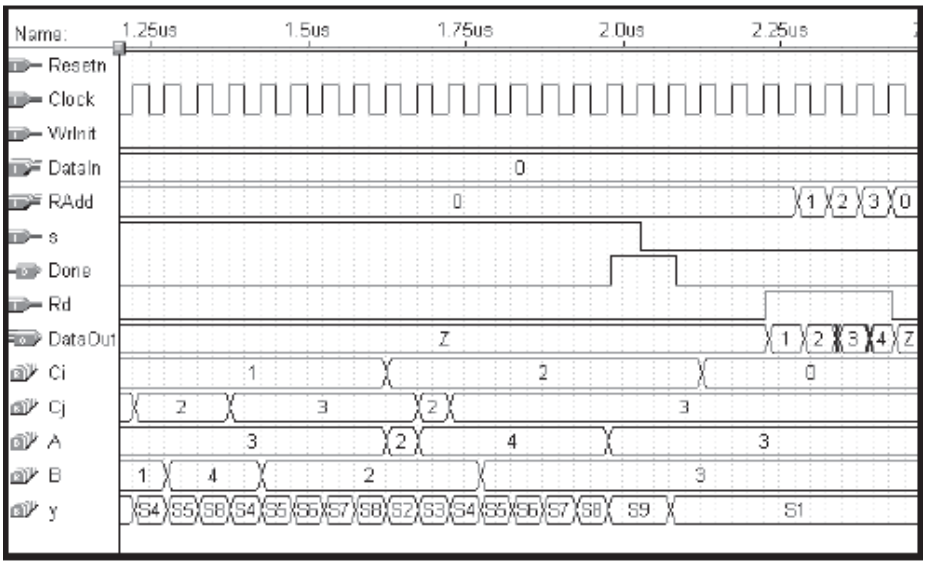
The FSM that controls the sort operation is described in the same way as in previous examples, using the processes *FSM_transitions* and *FSM_outputs*.

Following these processes, the code instantiates the registers R_0 to R_3, as well as *A* and *B*. The counters C_i and C_j are instantiated by the two statements labeled *OuterLoop* and *InnerLoop*, respectively. The multiplexers with the outputs *CMux* and *IMux* are specified using conditional signal assignments. The 4-to-1 multiplexer in Figure 7.42 is defined by the selected signal assignment that specifies the value of the *ABData* signal for each value of *IMux*. The 2-to-4 decoder in Figure 7.43 with the outputs Rin_0, …, Rin_3 is defined by the process statement labeled *RinDec*. Finally, the z_i and z_j signals and the *DataOut* output are specified using conditional signal assignments.

Figure 7.46 gives an example of a simulation result for the sorting example. Part (*a*) of the figure shows the first half of the simulation, from 0 to 1.25 µs, and part (*b*) shows the second half, from 1.25 µs to 2.5 µs. After resetting the circuit, *WrInit* is set to 1 for four clock cycles, and unsorted data is written into the four registers using the *DataIn* and *RAdd* inputs. After *s* is changed to 1, the FSM changes to state *S*2. States *S*2 to *S*4 load *A* with the contents of R_0 (3) and *B* with the contents of R_1 (2). State *S*5 compares *B* with *A*, and since *B*<*A*, the FSM uses states *S*6 and *S*7 to swap the contents of registers R_0 and R_1. In state *S*8, *A* is reloaded from R_0, which now contains 2. Since z_j is not asserted, the FSM increments the counter C_j and changes back to state *S*4. Register *B* is now loaded with the contents of R_2 (4), and the FSM changes to state *S*5. Since *B*=4 is not less than *A*=2, the machine changes to *S*8 and then back to *S*4. Register *B* is now loaded with the contents of R_3 (1), which is then compared against *A*=2 in state *S*5. The contents of R_0 and R_3 are swapped, and the machine changes to *S*8. At this point, the register contents are R_0=1, R_1=3, R_2=4, and R_3=2. Since z_j=1 and z_i=0, the FSM performs the next iteration of the outer loop by changing to state *S*2. Jumping forward in the simulation time, in Figure 7.46*b* the circuit reaches the state in which C_i=2, C_j=3, and the FSM is in state *S*8. The FSM then changes to state *S*9 and sets *Done* to the value 1. The correctly sorted data is read out of the registers by setting the signal *Rd* =1 and using the *RAdd* inputs to select each of the registers.



(a) Loading the registers and starting the sort operation



(b) Completing the sort operation and reading the registers

Figure 7.46 Simulation results for the sort operation.

7.8 Clock Synchronization and Timing Issues

In previous sections we presented several examples of logic circuits that illustrate how a designer may approach the task of designing larger systems. In this section

we will examine some timing aspects of such systems.

7.8.1 CLOCK DISTRIBUTION

In Chapter 5 we discussed the need to deal with the clock skew problem. For proper operation of synchronous sequential circuits, it is essential to minimize the clock skew as much as possible. Chips that contain many flip-flops use carefully designed networks of wires to distribute the clock signal to the flip-flops. Figure 7.47 gives an example of a clock-distribution network. Each node labeled *ff* represents the clock input of a flip-flop; for clarity, the flip-flops are not shown. The driver on the left of the figure produces the clock signal. This signal is distributed to the flip-flops such that the length of the wire between each flip-flop and the clock source is the same. Due to the appearance of sections of the wires, which resemble the letter H, the clock distribution network is known as an *H tree*. In chips the term *global clock* refers to the clock network. A chip usually provides one or more global clocks that can be connected to all flip-flops. When designing a circuit to be implemented in such a chip, a good design practice is to connect all the flip-flops in the circuit to a single global clock.

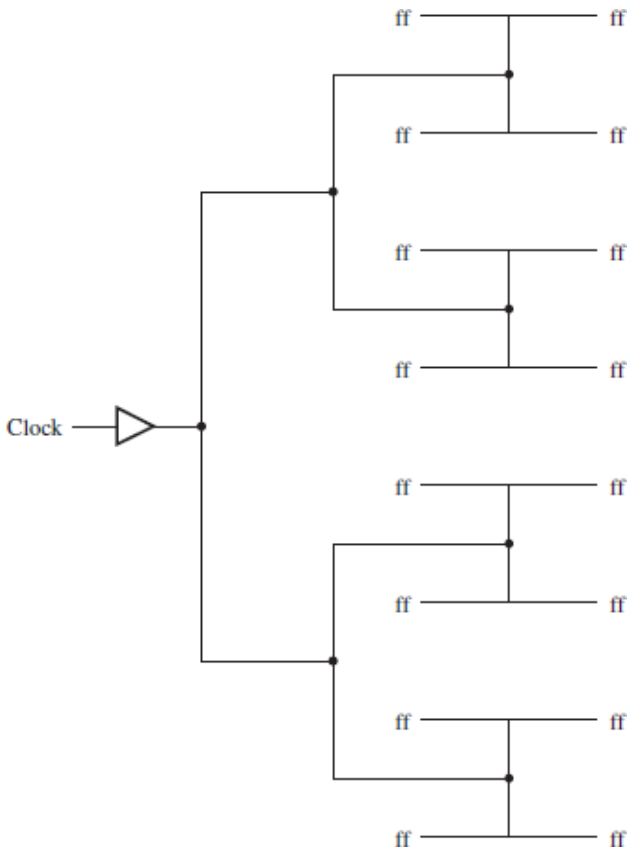


Figure 7.47 An H tree clock distribution network.

It is useful to be able to ensure that a sequential circuit is reset into a known state when power is first applied to the circuit. A good design practice is to connect the asynchronous reset (clear) inputs of all flip-flops to a wiring network that provides a low-skew reset signal. Chips usually provide a *global reset* wiring network for this purpose.

7.8.2 FLIP-FLOP TIMING PARAMETERS

We discussed the timing parameters for storage elements in Chapter 5. Data to be clocked into a flip-flop must be stable t_{su} before the active clock edge and must remain stable t_h after the clock edge. A change in the value of the output Q appears after the *clock-to*-Q *delay*, t_{cQ}. An *output delay time*, t_{od}, is required for the change in Q to propagate to an output pin on the chip. These timing parameters account for the behavior of an individual flip-flop without considering how the flip-flop is connected to other circuitry in an integrated circuit chip.

Figure 7.48 depicts a flip-flop as part of an integrated circuit. Connections are shown from the flip-flop's clock, *D*, and Q terminals to pins on the chip package. There is an input driver associated with each pin on the chip. Other circuitry may also be connected to the flip-flop; the shaded box represents a combinational circuit connected to *D*. The propagation delays between the pins on the chip package and the flip-flop are labeled in the figure as t_{Data}, t_{Clock}, and t_{od}.

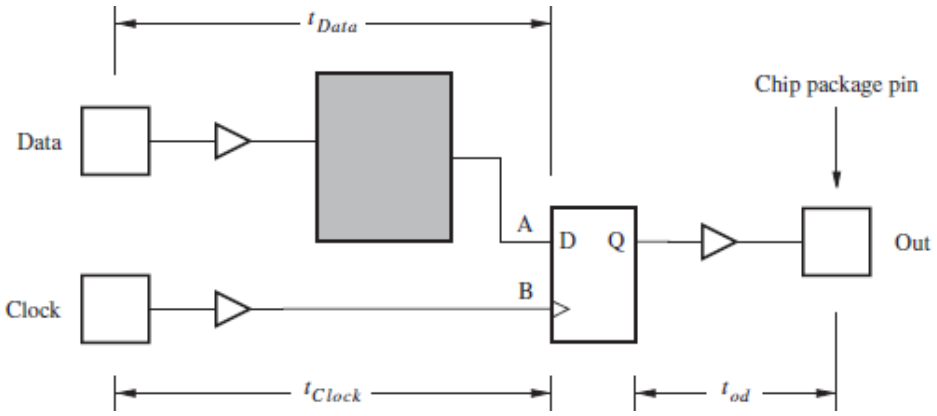


Figure 7.48 A flip-flop in an integrated circuit chip.

In digital systems the output signals from one chip are used as the input signals to another chip. Often the flip-flops in all chips are driven by a common clock that has low skew. The signals must propagate from the Q outputs of flip-flops in one chip to the *D* inputs of flip-flops in another chip. To ensure that all timing specifications are met, it is necessary to consider the output delays in one chip and the input delays in another.

The t_{co} delay determines how long it takes from when an active clock edge occurs at the clock pin on the chip package until a change in the output of a flip-flop appears at an output pin on the chip. This delay consists of three main parts. The clock signal must first propagate from its input pin on the chip to the flip-flop's *Clock* input. This delay is labeled t_{Clock} in Figure 7.48. After the clock-to-Q delay t_{cQ}, the flip-flop produces a new output, which takes t_{od} to propagate to the output pin. An example of timing parameters taken from a commercial chip is t_{Clock} = 1.5 ns, t_{cQ} = 1 ns, and t_{od} = 2 ns. These parameters give the delay from the active clock edge to the change on the output pin as t_{co} = 4.5 ns.

If chips are separated by a large distance, the propagation delays between them must be taken into consideration. But in most cases the distance between chips is small, and the propagation time of signals between the chips is negligible. Once a signal reaches the input pin on a chip, the relative values of t_{Data} and t_{Clock} (see Figure 7.48) must be considered. For example, in Figure 7.49 we assume that t_{Data} = 4.5 ns and t_{Clock} = 1.5 ns. The setup time for the flip-flops in the chip is specified as t_{su} = 3 ns. In the figure the *Data* signal changes from low to high 3 ns before the positive clock edge, which should meet the setup requirements. The *Data* signal takes 4.5 ns to reach the flip-flop, whereas the *Clock* signal takes only 1.5 ns. The signal labeled *A* and the clock signal labeled *B* reach the flip-flop at the same time. The setup time requirement is violated, and the flip-flop may become unstable. To avoid this condition, it is necessary to increase the setup time as seen from outside the chip.

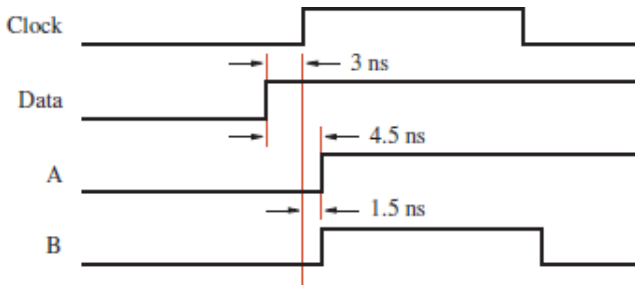


Figure 7.49 Flip-flop timing in a chip.

The hold time for flip-flops is also affected by chip-level delays. The result is usually a reduction in the hold time, rather than an increase. For example, with the timing parameters in Figure 7.49 assume that the hold time is t_h = 2 ns. Assume that the signal at the *Data* pin on the chip changes value at exactly the same time that an active edge occurs at the *Clock* pin. The change in the *Clock* signal will reach node *B* 4.5 - 1.5 = 3 ns before the change in *Data* reaches node

A. Hence even though the external change in *Data* is coincident with the clock edge, the required hold time of 2 ns is not violated.

For large circuits, ensuring that flip-flop timing parameters are properly adhered to is a challenge. Both the timing parameters of the flip-flops themselves and the relative delays incurred by the clock and data signals must be considered. CAD systems provide tools that can check the setup and hold times at all flip-flops automatically. This task is done using timing simulation, as well as special-purpose timing-analysis tools.

7.8.3 ASYNCHRONOUS INPUTS TO FLIP-FLOPS

In our examples of synchronous sequential circuits, we have assumed that changes in all input signals occur shortly after an active clock edge. The rationale for this assumption is that the inputs to one circuit are produced as the outputs of another circuit, and the same clock signal is used for both circuits. In practice, some of the inputs to a circuit may be generated asynchronously with respect to the clock signal. If these signals are connected to the *D* input of a flip-flop, then the setup or hold times may be violated.

When a flip-flop's setup or hold times are violated, the flip-flop's output may assume a voltage level that does not correspond to either logic value 0 or 1. We say that the flip-flop is in a *metastable* state. The flip-flop eventually settles in one of the stable states, 0 or 1, but the time required to recover from the metastable state is not predictable. A common approach for dealing with asynchronous inputs is illustrated in Figure 7.50. The asynchronous data input is connected to a two-bit shift register. The output of the first flip-flop, labeled *A* in the figure, will sometimes become metastable. But if the clock period is sufficiently long, then *A* will recover to a stable logic value before the next clock pulse occurs. Hence the output of the second flip-flop will not become metastable and can safely be connected to other parts of the circuit. The synchronization circuit introduces a delay of one clock cycle before the signal can be used by the rest of the circuit.

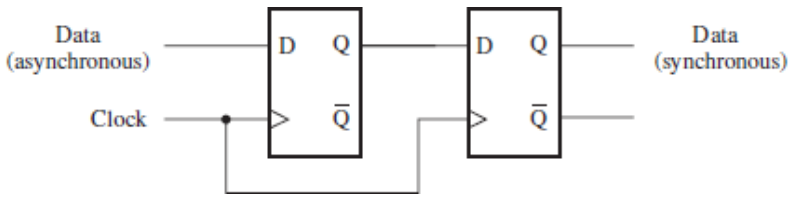


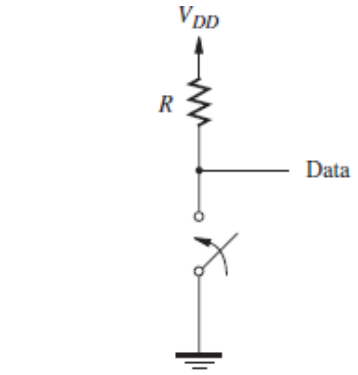
Figure 7.50 Asynchronous inputs.

Commercial chips specify the minimum allowable clock period that has to be used for the circuit in Figure 7.50 to solve the metastability problem. In practice,

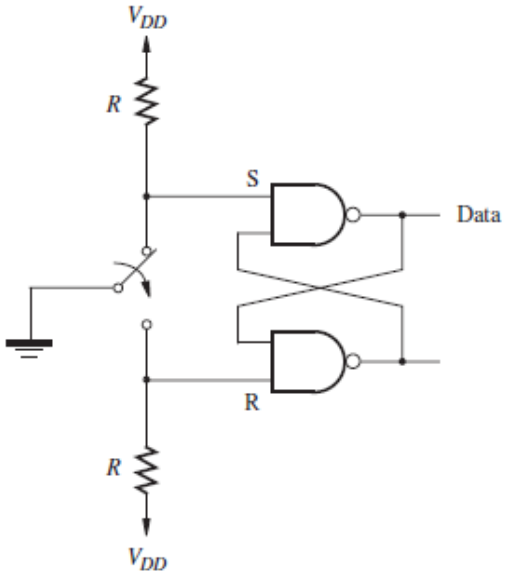
it is not possible to *guarantee* that node *A* will always be stable before a clock edge occurs. The data sheets specify a probability of node *A* being stable, as a function of the clock period. We will not pursue this issue further; the interested reader can refer to specialized books for a more detailed discussion.

7.8.4 SWITCH DEBOUNCING

Inputs to a logic circuit are sometimes generated by mechanical switches. A problem with such switches is that they bounce away from their contact points when changed from one position to the other. Figure 7.51*a* shows a single-pole single-throw switch that provides an input to a logic circuit. If the switch is open, then the *Data* signal has the value 1. When the switch is thrown to the closed position, *Data* becomes 0, but the switch bounces for some time, causing *Data* to oscillate between 1 and 0. The bouncing typically persists for about 10 ms.



(a) Single-pole single-throw switch



(b) Single-pole double-throw switch with a basic SR latch

Figure 7.51 Switch debouncing circuit.

One approach to deal with this problem is to add a capacitor, *C*, between the *Data* node and ground. When the switch is open, *C* is charged to the voltage V_{DD}. When the switch is thrown to the closed position, *C* is quickly discharged to 0 V (ground) through the switch which has a resistance close to zero ohm. Now, when the switch contact bounces away from its closed position, *C* will start to charge toward V_{DD} again. But, this charging will be slower due to the resistor *R*. By choosing the right sizes of *R* and *C*, it is possible to prevent the *Data* node from significantly changing its voltage during bouncing.

Another possible solution is to use a circuit, such as a counter, to measure an appropriately long delay to wait for the bouncing to stop (see Problem 7.20).

A good approach for dealing with switch bouncing is depicted in Figure 7.51*b*. It uses a single-pole double-throw switch and a basic SR latch to generate an input to a logic circuit. When the switch is in the bottom position, the *R* input on the latch is 0 and *Data* =0. When the switch is thrown to the top position, the *S* input on the latch becomes 0, which sets *Data* to 1. If the switch bounces away from the top position, the inputs to the latch become *R*=*S*=1 and the value *Data* =1 is stored by the latch. When the switch is thrown to the bottom position, *Data* changes to 0 and this value is stored in the latch if the switch bounces. Note that when a switch bounces, it cannot bounce fully between the *S* and *R* terminals; it only bounces slightly away from one of the terminals and then back to it.

7.9 Concluding Remarks

This chapter has provided several examples of digital systems that include one or more FSMs as well as building blocks like adders, registers, shift registers, and counters. We have shown how ASM charts can be used as an aid for designing a digital system, and we have shown how the circuits can be described using VHDL code. A number of practical issues have been discussed, such as clock distribution, synchronization of asynchronous inputs, and switch debouncing.

Problems

7.1 In Section 7.1 we showed a digital system with three registers, *R*1 to *R*3, and we designed a control circuit that can be used to swap the contents of registers *R*1 and *R*2. Give an ASM chart that represents this digital system and the swap operation.

7.2 (a) For the ASM chart derived in Problem 7.1, show another ASM chart that specifies the required control signals to control the datapath circuit. Assume that multiplexers are used to implement the bus that connects the registers, as shown in Figure 7.4.

(b) Write complete VHDL code for the system in Problem 7.1, including the control circuit described in part (*a*).

(c) Synthesize a circuit from the VHDL code written in part (*b*) and show a timing simulation that illustrates correct functionality of the circuit.

7.3 In Section 7.2 we designed a processor that performs the operations listed in Table 7.1. Design a modified circuit that performs an additional operation Swap *Rx*, *Ry*. This operation swaps the contents of registers *Rx* and *Ry*. Use three bits f_2, f_1, f_0 to represent the input *F* shown in Figure 7.11 because there are now five operations, rather than four. Add a new register, named *Tmp*, into the system, to be used for temporary storage during the swap operation. Show logic expressions for the outputs of the control circuit, as was done in Section 7.2.

7.4 In Section 7.2 we gave the design for a circuit that works as a processor. Give an ASM chart that describes the functionality of this processor.

7.5 (a) For the ASM chart derived in Problem 7.4, show another ASM chart that specifies the required control signals to control the datapath circuit in the processor. Assume that multiplexers are used to implement the bus that connects the registers, *R*0 to *R*3, in the processor.

(b) Write complete VHDL code for the system in Problem 7.4, including the control circuit described in part (*a*).

(c) Synthesize a circuit from the VHDL code written in part (*b*) and show a timing simulation that illustrates correct functionality of the circuit.

7.6 The ASM chart in Figure 7.17, which describes the bit-counting circuit, includes Moore-type outputs in states *S*1, *S*2, and *S*3, and it has a Mealy-type output in state *S*2.

(a) Show how the ASM chart can be modified such that it has only Moore-type outputs in state *S*2.

(b) Give the ASM chart for the control circuit corresponding to part (*a*).

(c) Give VHDL code that represents the modified control circuit.

7.7 Figure 7.24 shows the datapath circuit for the shift-and-add multiplier. It uses a shift register for *B* so that b_0 can be used to decide whether or not *A* should be added to *P*. A different approach is to use a normal register to hold operand *B* and to use a counter and multiplexer to select bit b_i in each stage of the multiplication operation.

(a) Show the ASM chart that uses a normal register for *B*, instead of a shift register.

(b) Show the datapath circuit corresponding to part (*a*).

(c) Give the ASM chart for the control circuit corresponding to part (*b*).

(d) Give VHDL code that represents the multiplier circuit.

7.8 Write VHDL code for the divider circuit that has the datapath in Figure 7.30 and the control circuit represented by the ASM chart in Figure 7.31.

7.9 Section 7.5 shows how to implement the traditional long division that is done by "hand." A different approach for implementing integer division is to perform repeated subtraction as indicated in the pseudo-code in Figure P7.1.

(a) Give an ASM chart that represents the pseudo-code in Figure P7.1.
(b) Show the datapath circuit corresponding to part (*a*).
(c) Give the ASM chart for the control circuit corresponding to part (*b*).
(d) Give VHDL code that represents the divider circuit.
(e) Discuss the relative merits and drawbacks of your circuit in comparison with the circuit designed in Section 7.5.

```
Q = 0;
R = A;
while ((R − B) > 0) do
    R = R − B;
    Q = Q + 1;
end while;
```

Figure P7.1 Pseudo-code for integer division.

7.10 In the ASM chart in Figure 7.39, the two states *S*3 and *S*4 are used to compute the mean $M = Sum/k$. Show a modified ASM chart that combines states *S*3 and *S*4 into a single state, called *S*3.

7.11 Write VHDL code for the FSM represented by your ASM chart defined in Problem 7.10.

7.12 In the ASM chart in Figure 7.41, we specify the assignment $C_j \leftarrow C_i$ in state *S*2, and then in state *S*3 we increment C_j by 1. Is it possible to eliminate state *S*3 if the assignment $C_j \leftarrow C_i + 1$ is performed in *S*2? Explain any implications that this change has on the control and datapath circuits.

7.13 Figure 7.40 gives pseudo-code for the sorting operation in which the registers being sorted are indexed using variables *i* and *j*. In the ASM chart in Figure 7.41, variables *i* and *j* are implemented using the counters C_i and C_j. A different approach is to implement *i* and *j* using two shift registers.

(a) Redesign the circuit for the sorting operation using the shift registers instead of the counters to index registers $R_0, \ldots, R_3$.
(b) Give VHDL code for the circuit designed in part (*a*).
(c) Discuss the relative merits and drawbacks of your circuit in comparison with the circuit that uses the counters C_i and C_j.

7.14 Design a circuit that finds the log_2 of an operand that is stored in an n-bit register. Show all steps in the design process and state any assumptions made. Give VHDL code that describes your circuit.

7.15 The circuit designed in Section 7.6 uses an adder to compute the sum of the contents of the registers. The divider subcircuit used to compute $M = Sum/k$ also includes an adder. Show how the circuit can be redesigned so that it contains only a single adder subcircuit that is used both for the summation operation and the division operation. Show only the extra circuitry needed to connect to the adder; and explain its operation.

7.16 Give VHDL code for the circuit designed in Problem 7.15, including both the datapath and control circuits.

7.17 The pseudo-code for the sorting operation given in Figure 7.40 uses registers A and B to hold the contents of the registers being sorted. Show pseudo-code for the sorting operation that uses only register A to hold temporary data during the sorting operation. Give a corresponding ASM chart that represents the datapath and control circuits needed. Use multiplexers to interconnect the registers, in the style shown in Figure 7.42. Give a separate ASM chart that represents the control circuit.

7.18 Give VHDL code for the sorting circuit designed in Problem 7.17.

7.19 Consider the design of a circuit that controls the traffic lights at the intersection of two roads. The circuit generates the outputs $G1$, $Y1$, $R1$ and $G2$, $Y2$, $R2$. These outputs represent the states of the green, yellow, and red lights, respectively, on each road. A light is turned on if the corresponding output signal has the value 1. The lights have to be controlled in the following manner: when $G1$ is turned on it must remain on for a time period called t_1 and then be turned off. Turning off $G1$ must result in $Y1$ being immediately turned on; it should remain on for a time period called t_2 and then be turned off. When either $G1$ or $Y1$ is on, $R2$ must be on and $G2$ and $Y2$ must be off. Turning off $Y1$ must result in $G2$ being immediately turned on for the t_1 time period. When $G2$ is turned off, $Y2$ is turned on for the t_2 time period. Of course, when either $G2$ or $Y2$ is turned on, $R1$ must be turned on and $G1$ and $Y1$ must be off.

(a) Give an ASM chart that describes the traffic-light controller. Assume that two down-counters exist, one that is used to measure the t_1 delay and another that is used to measure t_2. Each counter has parallel-load and enable inputs. These inputs are used to load an appropriate value representing either the t_1 or t_2 delay and then allow the counter to count down to 0.

(b) Give an ASM chart for the control circuit for the traffic-light controller.

(c) Write complete VHDL code for the traffic-light controller, including the control circuit from part (*a*) and counters to represent t_1 and t_2. Use any convenient clock frequency to clock the circuit and assume convenient count values to represent t_1 and t_2. Give simulation results that illustrate the operation of your circuit.

7.20 Assume that you need to use a single-pole single-throw switch as shown in Figure 7.51*a*. Show how a counter can be used as a means of debouncing the *Data* signal produced by the switch. (*Hint:* Design an FSM that has *Data* as an input and produces the output *z*, which is the debounced version of *Data*. Assume that you have access to a *Clock* input signal with the frequency 102.4 kHz, which can be used as needed.)

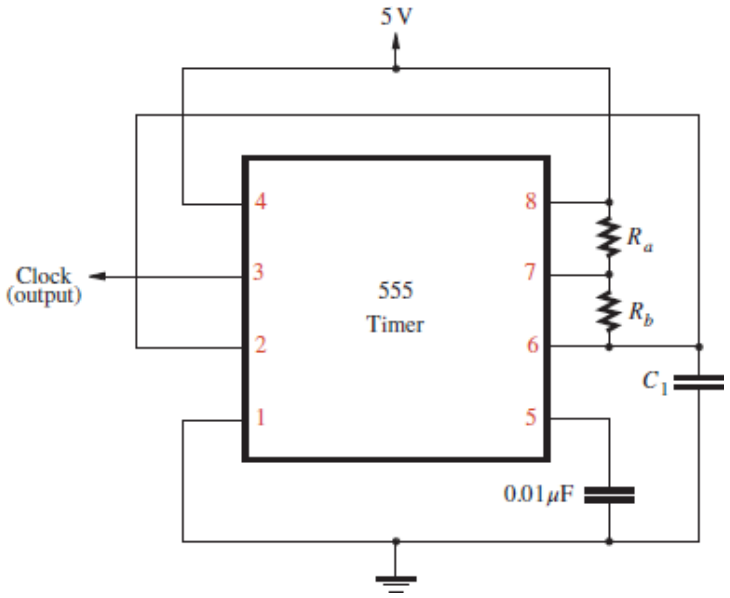


Figure P7.2 The 555 programmable timer chip.

7.21 Clock signals can be generated using special purpose chips. One example of such a chip is the 555 programmable timer, which is depicted in Figure P7.2. By choosing particular values for the resistors R_a and R_b and the capacitor C_1, the 555 timer can be used to produce a desired clock signal. It is possible to choose both the period of the clock signal and its duty cycle. The term *duty cycle* refers to the percentage of the clock period for which the signal is high. The following equations define the clock signal produced by the chip

$$\text{Clock period} = 0.7(R_a + 2R_b)C_1$$

$$\text{Duty cycle} = \frac{R_a + R_b}{R_a + 2R_b}$$

(a) Determine the values of R_a, R_b, and C_1 needed to produce a clock signal with a 50 percent duty cycle and a frequency of about 500 kHz.

(b) Repeat part (*a*) for a duty cycle of 75 percent.

chapter

8

OPTIMIZED IMPLEMENTATION OF LOGIC FUNCTIONS

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Synthesis of logic functions
- Analysis of logic circuits
- Algorithmic techniques for the optimization of logic functions
- Cubical representation of logic functions
- Representation of logic functions using binary decision diagrams

In Chapter 2 we showed how to find the lowest-cost implementations of logic functions by using either algebraic manipulation or Karnaugh maps. The reader is probably convinced that it is not at all obvious how to apply the theorems of Boolean algebra to find a minimum-cost circuit. Karnaugh

maps provide a systematic way of manually deriving minimum-cost implementations of simple logic functions, but they become impractical for functions of many variables.

If CAD tools are used to design logic circuits, the task of minimizing the cost of implementation does not fall to the designer; the tools perform the necessary optimizations automatically. Even so, it is important to know something about this process. Most CAD tools have many features and options that are under control of the user. To know when and how to apply these options, the user must have an understanding of what the tools do.

In this chapter we will introduce some of the optimization techniques implemented in CAD tools and show how these techniques can be automated. We first discuss synthesis techniques that can be used to produce multilevel logic circuits, rather than just the two-level sum-of-products and product-of-sums forms that we studied in Chapter 2. Then we will describe a cubical representation of logic functions, as well as a graphical representation called binary decision diagrams (BDDs). We will discuss some algorithmic approaches for circuit optimization that are based on these representations.

8.1 Multilevel Synthesis

In the previous chapters we have usually implemented logic functions in the sum-of-products or product-of-sums form. Logic circuits of this type have *two levels* of gates. In the sum-of-products form, the first level comprises AND gates that are connected to a second-level OR gate. In the product-of-sums form, the first-level OR gates feed the second-level AND gate. We have assumed that both true and complemented versions of the input variables are available so that NOT gates are not needed to complement the variables.

A two-level realization is usually efficient for functions of a few variables. However, as the number of inputs increases, a two-level circuit may result in logic gates that have too many inputs. Whether or not this is an issue depends on the type of technology that is used to implement the circuit. For example, consider the following function:

$$f(x_1, \ldots, x_7) = x_1 x_3 \bar{x}_6 + x_1 x_4 x_5 \bar{x}_6 + x_2 x_3 x_7 + x_2 x_4 x_5 x_7$$

This is a minimum-cost SOP expression. Now consider implementing f in two types of chips: a CPLD and an FPGA. Figure 8.1 shows a logic element in a typical CPLD, like those shown in Appendix B. The figure indicates in red the circuitry used to realize the function f. The CPLD architecture is well-suited for SOP implementations.

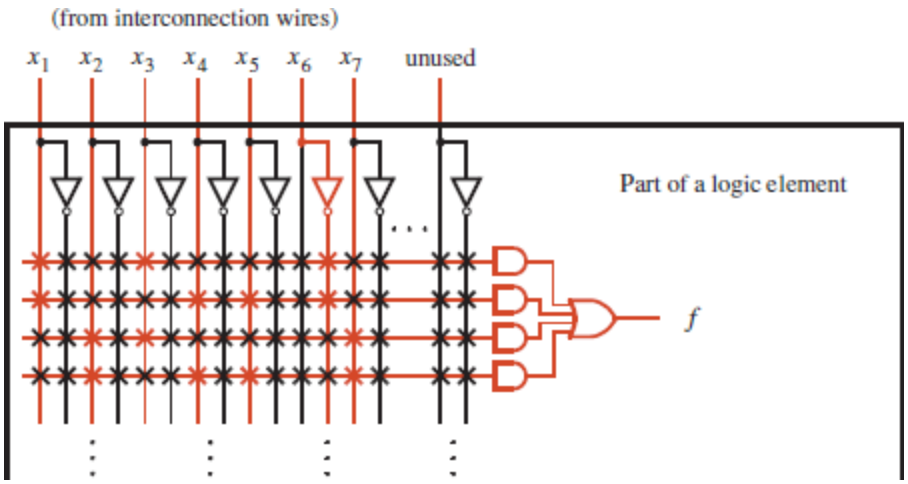


Figure 8.1 Implementation in a CPLD.

Next, consider implementing f in an FPGA. For this example we will assume an FPGA in which the logic elements are four-input lookup tables (LUTs). As discussed in Appendix B, a four-input LUT can implement any logic function that has up to four inputs. Since the SOP expression for f requires three- and four-input AND operations and a four-input OR, each of these logic operations could be realized in a separate LUT. Four LUTs would be needed for the AND operations, plus one additional LUT for the OR operation, giving five LUTs in total.

A more efficient implementation in the FPGA architecture can be obtained if f is expressed in a form that has more than two levels of logic operations. Such a form is called a *multilevel* logic expression. There are several different approaches for synthesis of multilevel circuits. We will discuss two important techniques known as *factoring* and *functional decomposition*.

8.1.1 FACTORING

The distributive property of Boolean algebra allows us to factor the preceding expression for f as follows

$$f = x_1\bar{x}_6(x_3 + x_4x_5) + x_2x_7(x_3 + x_4x_5)$$
$$= (x_1\bar{x}_6 + x_2x_7)(x_3 + x_4x_5)$$

The corresponding circuit can be realized using only two LUTs, as shown in Figure 8.2. The first LUT implements the expression $g = (x_1\bar{x}_6 + x_2x_7)$, and the second LUT implements $f = g \cdot (x_3 + x_4x_5)$.

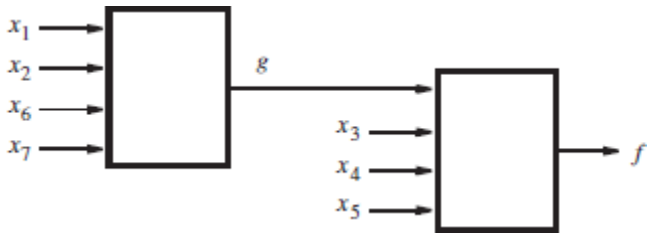


Figure 8.2 Using three LUTs after factoring.

Fan-in Problem

In the preceding example, the number of inputs, or fan-in, of each LUT is restricted to four because of the FPGA architecture. But even if we were to make a custom chip to implement a circuit it would still be necessary to limit the fan-in of gates. This limit is caused by practical issues related to transistor technology, which is discussed in Appendix B. Suppose that the available AND and OR gates in a custom chip have a maximum fan-in of four. Then, if we need to implement a logic expression that has (for example) a seven-input product term, then we would have to use 2 four-input AND gates, as indicated in Figure 8.3.

Figure 8.3 Using four-input AND gates to realize a seven-input product term.

Factoring can be used to deal with the fan-in problem. As an example, consider the function

$$f = x_1\bar{x}_2x_3\bar{x}_4x_5x_6 + x_1x_2\bar{x}_3\bar{x}_4\bar{x}_5x_6$$

This is a minimal sum-of-products expression. Using the approach of Figure 8.3, we will need four AND gates and one OR gate to implement this expression. Another solution is to factor the expression as follows

$$f = x_1\bar{x}_4x_6(\bar{x}_2x_3x_5 + x_2\bar{x}_3\bar{x}_5)$$

Then three AND gates and one OR gate suffice for realization of the required function, as depicted in Figure 8.4.

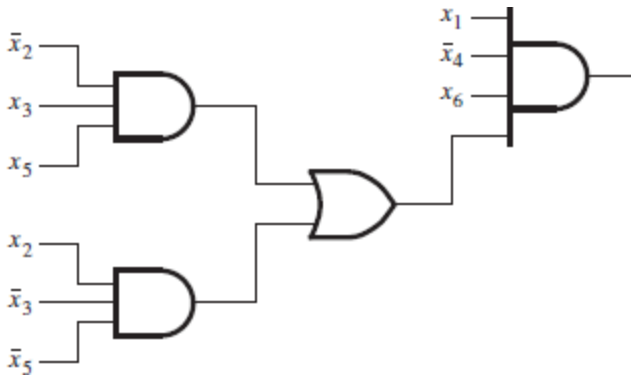


Figure 8.4 A factored circuit.

Example 8.1

In practical situations a designer of logic circuits often encounters specifications that naturally lead to an initial design where the logic expressions are in a factored form. Suppose we need a circuit that meets the following requirements. There are four inputs: x_1, x_2, x_3, and x_4. An output, f_1, must have the value 1 if at least one of the inputs x_1 and x_2 is equal to 1 and both x_3 and x_4 are equal to 1; it must also be 1 if $x_1 = x_2 = 0$ and either x_3 or x_4 is 1. In all other cases $f_1 = 0$. A different output, f_2, is to be equal to 1 in all cases except when both x_1 and x_2 are equal to 0 or when both x_3 and x_4 are equal to 0.

From this specification, the function f_1 can be expressed as

$$f_1 = (x_1 + x_2)x_3x_4 + \bar{x}_1\bar{x}_2(x_3 + x_4)$$

This expression can be simplified to

$$f_1 = x_3x_4 + \bar{x}_1\bar{x}_2(x_3 + x_4)$$

which the reader can verify by using a Karnaugh map.

The second function, f_2, is most easily defined in terms of its complement, such that

$$\bar{f}_2 = \bar{x}_1\bar{x}_2 + \bar{x}_3\bar{x}_4$$

Then using DeMorgan's theorem gives

$$f_2 = (x_1 + x_2)(x_3 + x_4)$$

which is the minimum-cost expression for f_2; the cost increases significantly if the SOP form is used.

Because our objective is to design the lowest-cost combined circuit that implements f_1 and f_2, it seems that the best result can be achieved if we use the factored forms for both functions, in which case the sum term $(x_3 + x_4)$ can be shared. Moreover, observing that $\bar{x}_1\bar{x}_2 = \overline{x_1 + x_2}$, the sum term $(x_1 + x_2)$ can also be shared if we express f_1 in the form

$$f_1 = x_3x_4 + \overline{x_1 + x_2}(x_3 + x_4)$$

Then the combined circuit, shown in Figure 8.5, comprises three OR gates, three AND gates, one NOT gate, and 13 inputs, for a total cost of 20. This cost includes the total number of gates in the circuit (7) plus the total number of gate-inputs (13).

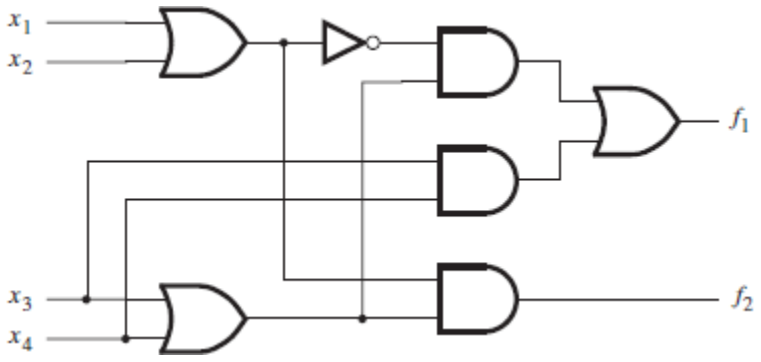


Figure 8.5 Circuit for Example 8.1.

8.1.2 FUNCTIONAL DECOMPOSITION

In the preceding examples, which illustrate the factoring approach, multilevel circuits are used to deal with fan-in limitations. However, such circuits may be preferable to their two-level equivalents even if fan-in is not a problem. In some cases the multilevel circuits may reduce the cost of implementation. On the other hand, they usually imply longer propagation delays, because they use multiple stages of logic gates. We will explore these issues by means of illustrative examples.

Complexity of a logic circuit can often be reduced by *decomposing* a two-level circuit into subcircuits, where one or more subcircuits implement functions that may be used in several places to construct the final circuit. To achieve this objective, a two-level logic expression is replaced by two or more new expressions, which are then combined to define a multilevel circuit. We can illustrate this idea by a simple example.

Example 8.2

Consider the four-variable function f shown in the Karnaugh map in Figure 8.6*a*. The map indicates how the function f can be realized in minimum SOP form, leading to the expression

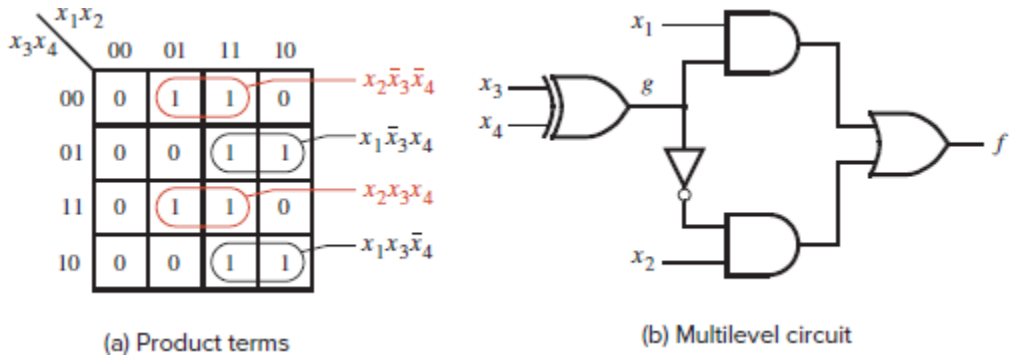


Figure 8.6 The function for Example 8.2.

$$f = x_1\bar{x}_3x_4 + x_1x_3\bar{x}_4 + x_2\bar{x}_3\bar{x}_4 + x_2x_3x_4$$

This expression has five gates and 16 inputs to gates, for a total cost of 21. We have assumed that the primary inputs are available in both their true and

complemented forms, and have therefore not included the cost of NOT gates needed to complement x_3 and x_4.

To perform functional decomposition we need to find one or more subfunctions that depend on only a subset of the input variables. In this example we will attempt to find a subfunction that depends only on the inputs x_3 and x_4. Factoring x_1 from the first two terms and x_2 from the last two terms, we have

$$f = x_1(\bar{x}_3x_4 + x_3\bar{x}_4) + x_2(\bar{x}_3\bar{x}_4 + x_3x_4)$$

Now let $g(x_3, x_4) = \bar{x}_3x_4 + x_3\bar{x}_4$ and observe that $\bar{g} = \bar{x}_3\bar{x}_4 + x_3x_4$. The decomposed function *f* can be written as

$$f = x_1g + x_2\bar{g}$$

Observing that $g = x_3 \oplus x_4$ leads to the circuit shown in Figure 8.6*b*. It has five logic gates plus nine inputs to gates, leading to a total cost of 13. The cost of this multilevel circuit is significantly lower than that of the SOP implementation of *f*. The trade-off is an increased propagation delay because the circuit has an extra level of logic.

Instead of using algebraic manipulation to find subfunctions, a more systematic approach can be followed by examining the Karnaugh map. Figure 8.7*a* highlights subfunction *g* in red. This subfunction depends only on the variables that define the rows of the Karnaugh map, namely x_3 and x_4. We have

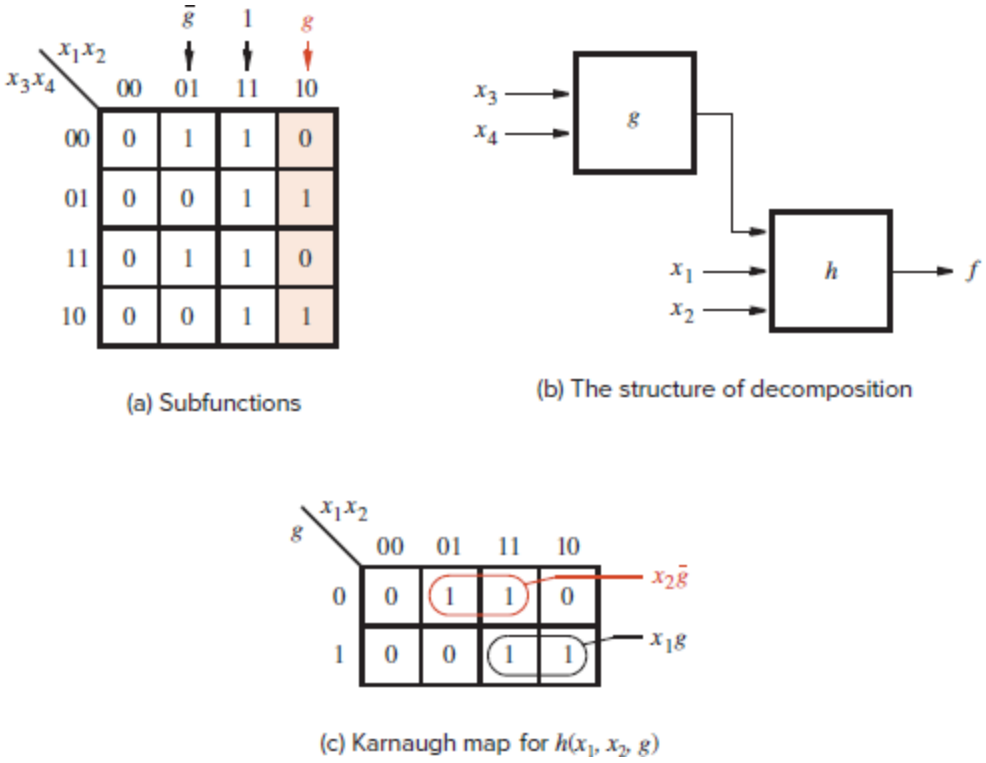


Figure 8.7 Subfunctions used in decomposition.

$$g = \bar{x}_3x_4 + x_3\bar{x}_4$$
$$= x_3 \oplus x_4$$

Since g is in the rightmost column of the map, where $x_1 = 1$ and $x_2 = 0$, the part of f defined by g is given by $(x_1\bar{x}_2) \cdot g$. Now, we can observe that the second column in the Karnaugh map represents the subfunction $\bar{g}$, and the third column is a subfunction equal to 1. Thus, f can be implemented using a three-variable function $h(x_1, x_2, g)$ such that

$$f(x_1, x_2, x_3, x_4) = h[x_1, x_2, g(x_3, x_4)]$$
$$= (\bar{x}_1x_2) \cdot \bar{g} + (x_1\bar{x}_2) \cdot g + (x_1x_2) \cdot 1$$

The structure of this decomposition is illustrated in Figure 8.7*b*. We can optimize the expression for h as shown in Figure 8.7*c*. The result is

$$h = x_1g + x_2\bar{g}$$

which is the same decomposition that we derived above by using algebraic manipulation.

In Figure 8.7*a* we identified subfunctions by first looking at the columns of the map. Decomposition can also be derived by first examining the rows of a map, which is the approach that we explore in the following example.

Example 8.3

Figure 8.8*a* defines a five-variable function f in the form of a Karnaugh map. Examining the rows of the map, we can see that there are only two distinct patterns, which means that only two subfunctions are needed to implement the rows. The second and fourth rows have one pattern, highlighted in red, while the first and third rows have the other pattern. Consider the second row. It depends only on the variables that define columns in each row, namely, x_1, x_2, and x_5. Let a subfunction $g(x_1, x_2, x_5)$ represent the pattern in this row. This subfunction is just

$$g = x_1 + x_2 + x_5$$

because the pattern has a 1 wherever any of these variables is equal to 1. To specify the location of rows where the pattern g occurs, which is in rows 2 and 4, we use the variables x_3 and x_4. The terms $\bar{x}_3 x_4$ and $x_3 \bar{x}_4$ identify the second and fourth rows, respectively. Thus the expression $(\bar{x}_3 x_4 + x_3 \bar{x}_4) \cdot g$ represents the part of f that is defined in rows 2 and 4.

Next, we have to find a realization for the pattern in rows 1 and 3. This pattern has a 1 only in the cell where $x_1 = x_2 = x_5 = 0$, which corresponds to the term $\bar{x}_1 \bar{x}_2 \bar{x}_5$. But we can make a useful observation that this term is just a complement of g. The location of rows 1 and 3 is identified by terms $\bar{x}_3 \bar{x}_4$ and $x_3 x_4$, respectively. Thus the expression $(\bar{x}_3 \bar{x}_4 + x_3 x_4) \cdot \bar{g}$ represents f in rows 1 and 3.

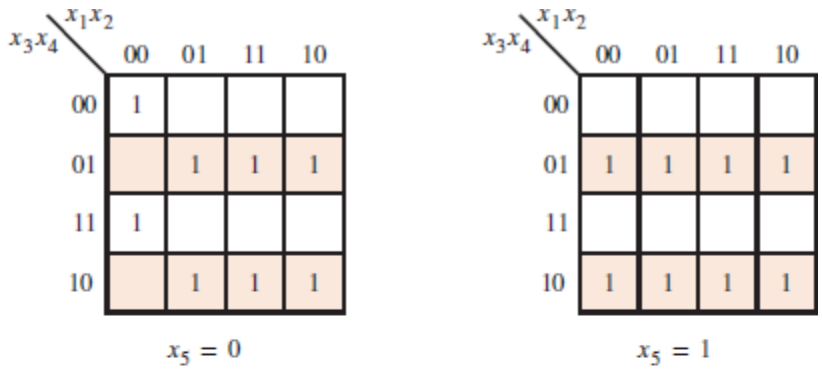
We can make one other useful observation. The expressions $(\bar{x}_3 x_4 + x_3 \bar{x}_4)$ and $(\bar{x}_3 \bar{x}_4 + x_3 x_4)$ are complements of each other. Therefore, if we let $k(x_3, x_4) = \bar{x}_3 x_4 + x_3 \bar{x}_4$, the complete decomposition of f can be stated as

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) &= h[g(x_1, x_2, x_5), k(x_3, x_4)] \\ &= kg + \bar{k}\bar{g} = \overline{k \oplus g} \end{aligned}$$

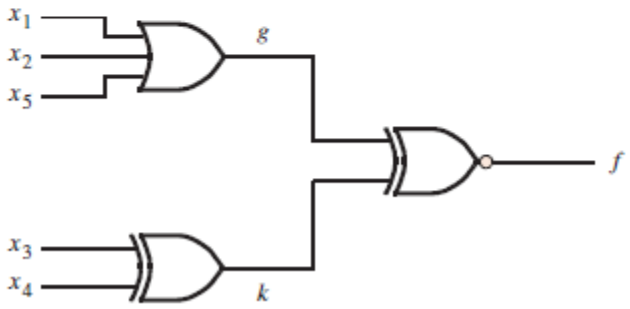
where

$$\begin{aligned} g &= x_1 + x_2 + x_5 \\ k &= \bar{x}_3 x_4 + x_3 \bar{x}_4 = x_3 \oplus x_4 \end{aligned}$$

The resulting circuit is given in Figure 8.8*b*. It requires a total of three gates and seven inputs to gates, for a total cost of 10. The largest fan-in of any gate is three.



(a) Karnaugh map for the function f



(b) Circuit obtained using decomposition

Figure 8.8 Decomposition for Example 8.3.

For comparison, a minimum-cost sum-of-products expression for f is

$$f = x_1\bar{x}_3x_4 + x_1x_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_2x_3\bar{x}_4 + \bar{x}_3x_4x_5 + x_3\bar{x}_4x_5 + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5 + \bar{x}_1\bar{x}_2x_3x_4\bar{x}_5$$

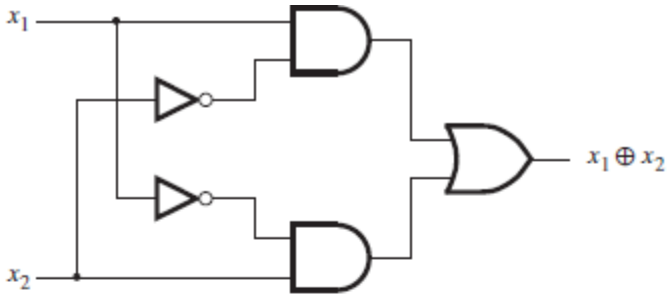
The corresponding circuit requires a total of 9 gates (excluding the five NOT gates to complement the primary inputs) and 36 inputs, giving a total cost of 45. The fan-in for the output OR gate is eight. Obviously, functional decomposition results in a much simpler implementation of this function.

In the two preceding examples, the decomposition is such that a decomposed subfunction depends on some primary input variables, whereas the remainder of the implementation depends on the rest of the variables.

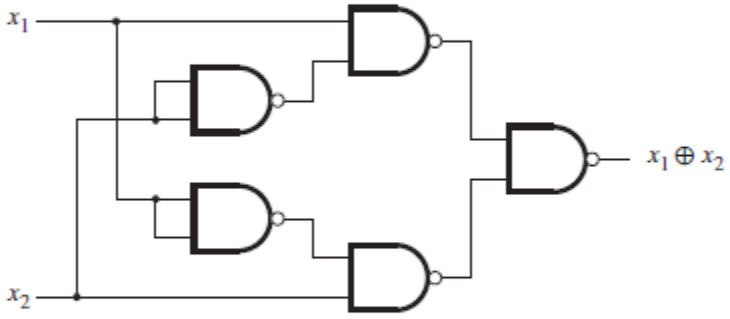
Such decompositions are called *disjoint decompositions* in the technical literature. It is possible to have a *non-disjoint decomposition*, where the variables of the subfunction are also used in realizing the remainder of the circuit. The following example illustrates this possibility.

Example 8.4

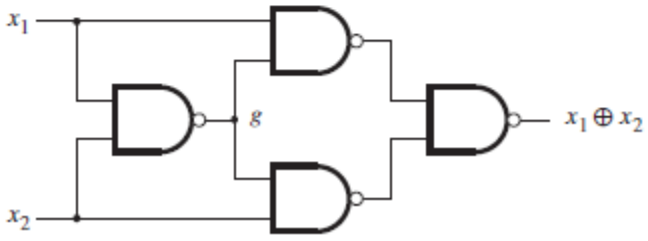
Suppose that we wish to implement the XOR function using only NAND gates. Figure 8.9*a* shows a SOP implementation of this function, and part (*b*) of the figure gives a NAND-NAND circuit that has the same structure.



(a) Sum-of-products implementation



(b) NAND gate implementation



(c) Optimal NAND gate implementation

Figure 8.9 Implementation of XOR.

Let us now try to exploit functional decomposition to find a better implementation of XOR using only NAND gates. Let the symbol $\uparrow$ represent the NAND operation so that $x_1 \uparrow x_2 = \overline{x_1 \cdot x_2}$. A sum-of-products expression for the XOR function is

$$x_1 \oplus x_2 = x_1\bar{x}_2 + \bar{x}_1 x_2$$

From the discussion in Section 2.7, this expression can be written in terms of NAND operations as

$$x_1 \oplus x_2 = (x_1 \uparrow \bar{x}_2) \uparrow (\bar{x}_1 \uparrow x_2)$$

This expression requires five NAND gates, and it is implemented by the circuit in Figure 8.9*b*. Observe that an inverter is implemented using a two-input NAND gate by tying the two inputs together.

To find a decomposition, we can manipulate the term $(x_1 \uparrow \bar{x}_2)$ as follows:

$$(x_1 \uparrow \bar{x}_2) = \overline{(x_1\bar{x}_2)} = \overline{(x_1(\bar{x}_1 + \bar{x}_2))} = (x_1 \uparrow (\bar{x}_1 + \bar{x}_2))$$

We can perform a similar manipulation for $(\bar{x}_1 \uparrow x_2)$ to generate

$$x_1 \oplus x_2 = (x_1 \uparrow (\bar{x}_1 + \bar{x}_2)) \uparrow ((\bar{x}_1 + \bar{x}_2) \uparrow x_2)$$

DeMorgan's theorem states that $\bar{x}_1 + \bar{x}_2 = x_1 \uparrow x_2$; hence we can write

$$x_1 \oplus x_2 = (x_1 \uparrow (x_1 \uparrow x_2)) \uparrow ((x_1 \uparrow x_2) \uparrow x_2)$$

Now we have a decomposition

$$x_1 \oplus x_2 = (x_1 \uparrow g) \uparrow (g \uparrow x_2)$$
$$g = x_1 \uparrow x_2$$

The corresponding circuit, which requires only four NAND gates, is given in Figure 8.9*c*.

Practical Issues

Functional decomposition is a powerful technique for reducing the complexity of circuits. A big problem in functional decomposition is finding the possible subfunctions. For functions of many variables, an enormous number of possibilities should be tried. This situation precludes attempts at finding optimal solutions. Instead, heuristic approaches that lead to acceptable solutions are used.

Full discussion of functional decomposition and factoring is beyond the scope of this book. An interested reader may consult more specialized texts on the topic of logic synthesis or switching theory. Modern CAD tools use the concept of decomposition extensively.

8.1.3 MULTILEVEL NAND AND NOR CIRCUITS

In Section 2.7 we showed that two-level circuits consisting of AND and OR gates can be easily converted into circuits that can be realized with NAND and NOR gates, using the same gate arrangement. In particular, an AND-OR (sum-of-products) circuit can be realized as a NAND-NAND circuit, while an OR-AND (product-of-sums) circuit becomes a NOR-NOR circuit. The same conversion approach can be used for multilevel circuits. We will illustrate this approach by an example.

Example 8.5

Figure 8.10*a* gives a four-level circuit consisting of AND and OR gates. Let us first derive a functionally equivalent circuit that comprises only NAND gates. Each AND gate is converted to a NAND by inverting its output. Each OR gate is converted to a NAND by inverting its inputs. This is just an application of DeMorgan's theorem, as illustrated in Figure 2.26. Figure 8.10*b* shows the necessary inversions in red. Note that an inversion is applied at both ends of a given wire. Now each gate becomes a NAND gate. This accounts for most of the inversions added to the original circuit. But, there are still five inversions that are not a part of any gate; therefore, they must be implemented separately. These inversions are at inputs x_1, x_5, x_6, and x_7 and at the output f. They can be implemented as two-input NAND gates, where the inputs are tied together. The resulting circuit is shown in Figure 8.10*c*.

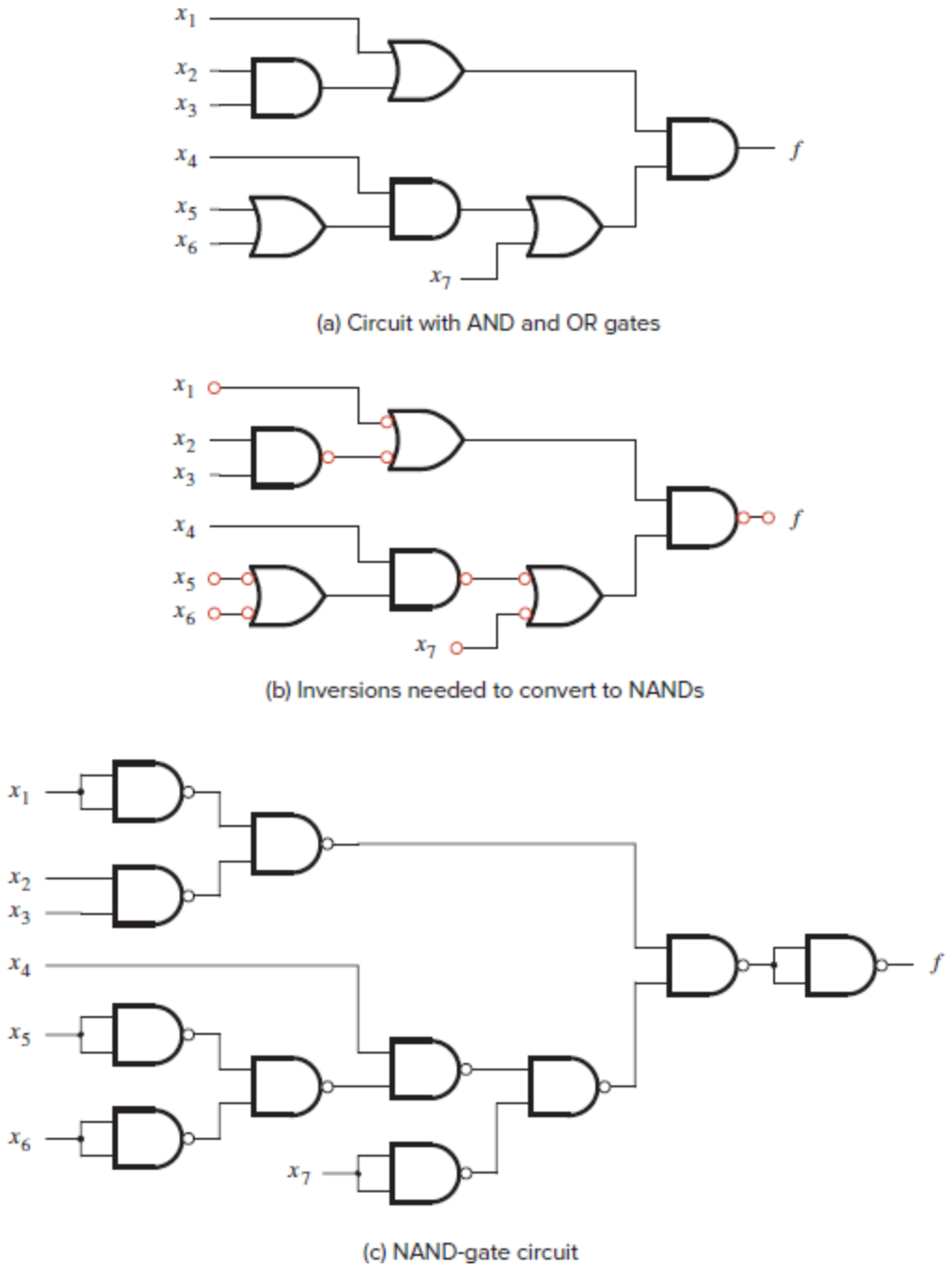
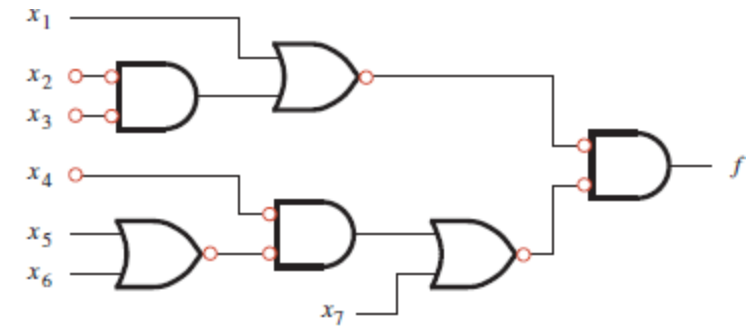


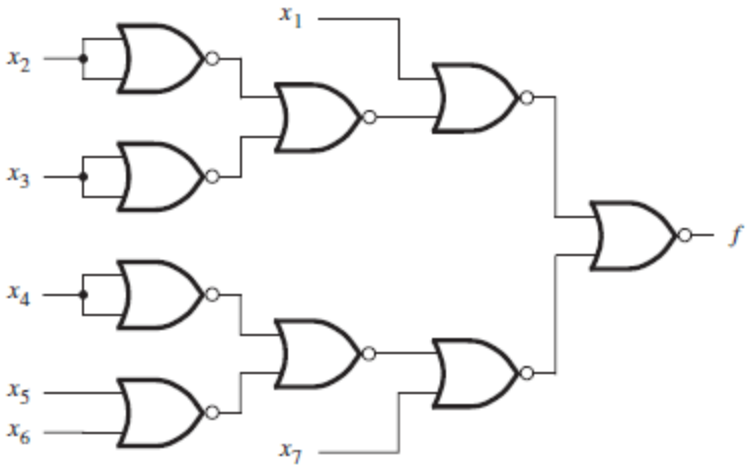
Figure 8.10 Conversion to a NAND-gate circuit.

A similar approach can be used to convert the circuit in Figure 8.10*a* into a circuit that comprises only NOR gates. An OR gate is converted to a NOR gate by inverting its output. An AND becomes a NOR if its inputs are inverted, as indicated in Figure 2.26. Using this approach, the inversions needed for our sample circuit are shown in red in Figure 8.11*a*. Then each

gate becomes a NOR gate. The three inversions at inputs x_2, x_3, and x_4 can be realized as two-input NOR gates, where the inputs are tied together. The resulting circuit is presented in Figure 8.11*b*.



(a) Inversions needed to convert to NORs



(b) NOR-gate circuit

Figure 8.11 Conversion to a NOR-gate circuit.

It is evident that the basic topology of a circuit does not change substantially when converting from AND and OR gates to either NAND or NOR gates. However, it may be necessary to insert additional gates to serve as NOT gates that implement inversions not absorbed as a part of other gates in the circuit.

8.2 Analysis of Multilevel Circuits

The preceding section showed that it may be advantageous to implement logic functions using multilevel circuits. It also presented the most commonly used approaches for synthesizing functions in this way. In this section we will consider the task of analyzing an existing circuit to determine the function that it implements.

For two-level circuits the analysis process is simple. If a circuit has an AND-OR (NAND- NAND) structure, then its output function can be written in the SOP form by inspection. Similarly, it is easy to derive a POS expression for an OR-AND (NOR-NOR) circuit. The analysis task is more complicated for multilevel circuits because it is difficult to write an expression for the function by inspection. We have to derive the desired expression by tracing the circuit and determining its functionality. The tracing can be done either starting from the input side and working towards the output, or by starting at the output side and working back towards the inputs. At intermediate points in the circuit, it is necessary to evaluate the subfunctions realized by the logic gates.

Example 8.6

Figure 8.12 replicates the circuit from Figure 8.10*a*. To determine the function f implemented by this circuit, we can consider the functionality at internal points that are the outputs of various gates. These points are labeled P_1 to P_5 in the figure. The functions realized at these points are

$$P_1 = x_2x_3$$
$$P_2 = x_5 + x_6$$
$$P_3 = x_1 + P_1 = x_1 + x_2x_3$$
$$P_4 = x_4P_2 = x_4(x_5 + x_6)$$
$$P_5 = P_4 + x_7 = x_4(x_5 + x_6) + x_7$$

Then f can be evaluated as

$$f = P_3P_5$$
$$= (x_1 + x_2x_3)(x_4(x_5 + x_6) + x_7)$$

Applying the distributive property to eliminate the parentheses gives

$$f = x_1x_4x_5 + x_1x_4x_6 + x_1x_7 + x_2x_3x_4x_5 + x_2x_3x_4x_6 + x_2x_3x_7$$

Note that the expression represents a circuit comprising six AND gates, one OR gate, and 25 inputs. The cost of this two-level circuit is higher than the cost of the circuit in Figure 8.12, but the circuit has lower propagation delay.

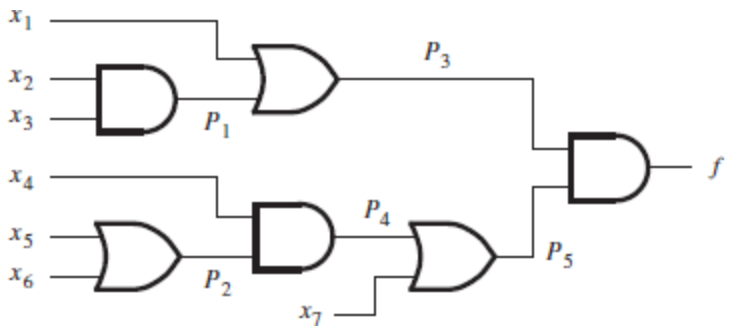


Figure 8.12 Circuit for Example 8.6.

Example 8.7

In Example 8.3 we derived the circuit in Figure 8.8*b*. This circuit is reproduced in Figure 8.13, except that we have replaced the XOR and XNOR gates from Figure 8.8*b* with their equivalent SOP form. The internal points in Figure 8.13 are labeled from P_1 to P_{10} as shown. The following subfunctions occur

$$P_1 = x_1 + x_2 + x_5$$
$$P_2 = \overline{x}_4$$
$$P_3 = \overline{x}_3$$
$$P_4 = x_3P_2$$
$$P_5 = x_4P_3$$
$$P_6 = P_4 + P_5$$
$$P_7 = \overline{P}_1$$
$$P_8 = \overline{P}_6$$
$$P_9 = P_1P_6$$
$$P_{10} = P_7P_8$$

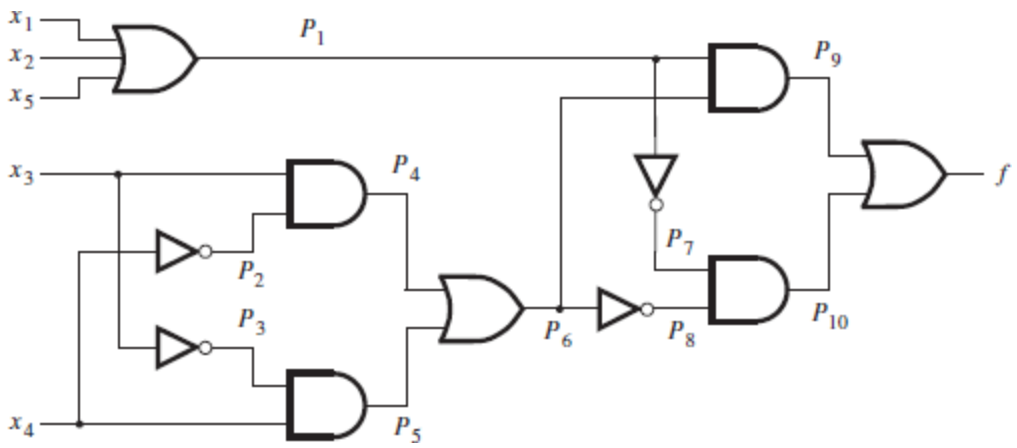


Figure 8.13 Circuit for Example 8.7.

We can derive f by tracing the circuit from the output towards the inputs as follows

$$
\begin{aligned}
f &= P_9 + P_{10} \\
&= P_1 P_6 + P_7 P_8 \\
&= (x_1 + x_2 + x_5)(P_4 + P_5) + \overline{P}_1 \overline{P}_6 \\
&= (x_1 + x_2 + x_5)(x_3 P_2 + x_4 P_3) + \overline{x}_1 \overline{x}_2 \overline{x}_5 \overline{P}_4 \overline{P}_5 \\
&= (x_1 + x_2 + x_5)(x_3 \overline{x}_4 + x_4 \overline{x}_3) + \overline{x}_1 \overline{x}_2 \overline{x}_5 (\overline{x}_3 + \overline{P}_2)(\overline{x}_4 + \overline{P}_3) \\
&= (x_1 + x_2 + x_5)(x_3 \overline{x}_4 + \overline{x}_3 x_4) + \overline{x}_1 \overline{x}_2 \overline{x}_5 (\overline{x}_3 + x_4)(\overline{x}_4 + x_3) \\
&= x_1 x_3 \overline{x}_4 + x_1 \overline{x}_3 x_4 + x_2 x_3 \overline{x}_4 + x_2 \overline{x}_3 x_4 + x_5 x_3 \overline{x}_4 + x_5 \overline{x}_3 x_4 \\
&\quad + \overline{x}_1 \overline{x}_2 \overline{x}_5 \overline{x}_3 \overline{x}_4 + \overline{x}_1 \overline{x}_2 \overline{x}_5 x_4 x_3
\end{aligned}
$$

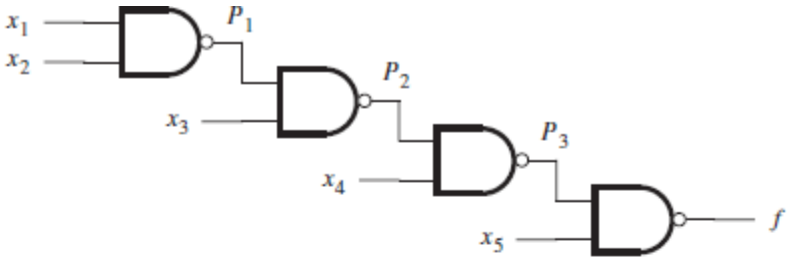
This is the same expression as stated in Example 8.3.

Example 8.8

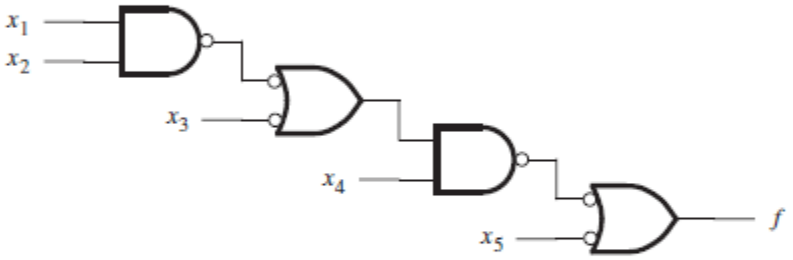
Circuits based on NAND and NOR gates are slightly more difficult to analyze because each gate involves an inversion. Figure 8.14*a* depicts a simple NAND-gate circuit that illustrates the effect of inversions. We can convert this circuit into a circuit with AND and OR gates using the reverse of the approach described in Example 8.5. Bubbles that denote inversions can be moved, according to DeMorgan's theorem, as indicated in Figure 8.14*b*. Then the circuit can be converted into the circuit in part (*c*) of the

figure, which consists of AND and OR gates. Observe that in the converted circuit, the inputs x_3 and x_5 are complemented. From this circuit the function f is determined as

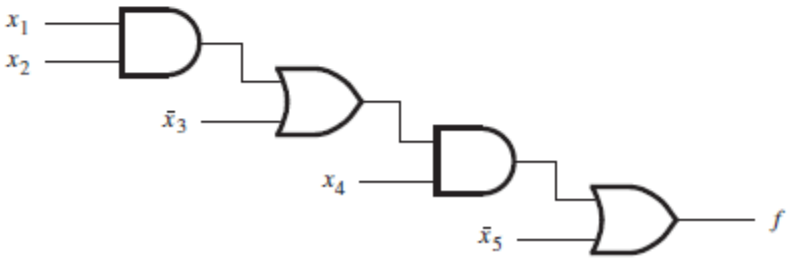
$$f = (x_1x_2 + \bar{x}_3)x_4 + \bar{x}_5$$
$$= x_1x_2x_4 + \bar{x}_3x_4 + \bar{x}_5$$



(a) NAND-gate circuit



(b) Moving bubbles to convert to ANDs and ORs



(c) Circuit with AND and OR gates

Figure 8.14 Circuit for Example 8.8.

It is not necessary to convert a NAND circuit into a circuit with AND and OR gates to determine its functionality. We can use the approach from Examples 8.6 and 8.7 to derive f as follows. Let P_1, P_2, and P_3 label the internal points as shown in Figure 8.14*a*. Then

$$
\begin{aligned}
P_1 &= \overline{x_1 x_2} \\
P_2 &= \overline{P_1 x_3} \\
P_3 &= \overline{P_2 x_4} \\
f &= \overline{P_3 x_5} = \overline{P}_3 + \overline{x}_5 \\
&= \overline{\overline{P_2 x_4}} + \overline{x}_5 = P_2 x_4 + \overline{x}_5 \\
&= \overline{P_1 x_3} x_4 + \overline{x}_5 = (\overline{P}_1 + \overline{x}_3) x_4 + \overline{x}_5 \\
&= (\overline{\overline{x_1 x_2}} + \overline{x}_3) x_4 + \overline{x}_5 \\
&= (x_1 x_2 + \overline{x}_3) x_4 + \overline{x}_5 \\
&= x_1 x_2 x_4 + \overline{x}_3 x_4 + \overline{x}_5
\end{aligned}
$$

Example 8.9

The circuit in Figure 8.15 consists of NAND and NOR gates. It can be analyzed as follows.

$$
\begin{aligned}
P_1 &= \overline{x_2 x_3} \\
P_2 &= \overline{x_1 P_1} = \overline{x}_1 + \overline{P}_1 \\
P_3 &= \overline{x_3 x_4} = \overline{x}_3 + \overline{x}_4 \\
P_4 &= \overline{P_2 + P_3} \\
f &= \overline{P_4 + x_5} = \overline{P}_4 \overline{x}_5 \\
&= \overline{\overline{P_2 + P_3} \cdot \overline{x}_5} \\
&= (P_2 + P_3)\overline{x}_5 \\
&= (\overline{x}_1 + \overline{P}_1 + \overline{x}_3 + \overline{x}_4)\overline{x}_5 \\
&= (\overline{x}_1 + x_2 x_3 + \overline{x}_3 + \overline{x}_4)\overline{x}_5 \\
&= (\overline{x}_1 + x_2 + \overline{x}_3 + \overline{x}_4)\overline{x}_5 \\
&= \overline{x}_1 \overline{x}_5 + x_2 \overline{x}_5 + \overline{x}_3 \overline{x}_5 + \overline{x}_4 \overline{x}_5
\end{aligned}
$$

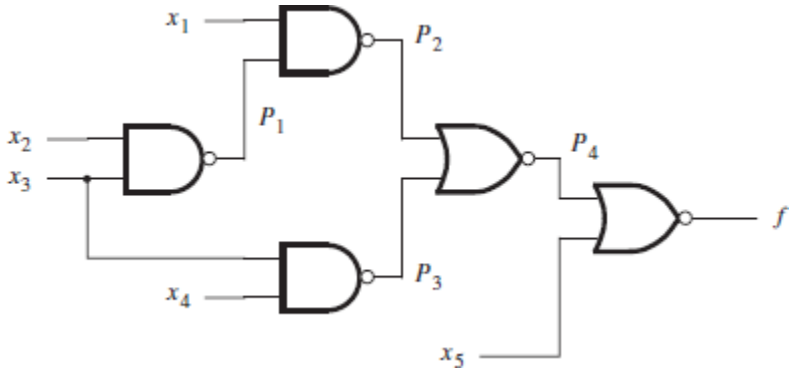


Figure 8.15 Circuit for Example 8.9.

Note that in deriving the second to the last line, we used property 16*a* in Section 2.5 to simplify $x_2x_3 + \bar{x}_3$ into $x_2 + \bar{x}_3$.

Analysis of circuits is much simpler than synthesis. With a little practice one can develop an ability to easily analyze even fairly complex circuits.

We have now covered a considerable amount of material on synthesis and analysis of logic functions. We have used the Karnaugh map as a vehicle for illustrating the concepts involved in finding optimal implementations of logic functions. We have also shown that logic functions can be realized in a variety of forms, both with two levels of logic and with multiple levels. In a modern design environment, logic circuits are synthesized using CAD tools, rather than by hand. The concepts that we have discussed in this chapter are quite general; they are representative of the strategies implemented in CAD algorithms. As we have said before, the Karnaugh map scheme for representing logic functions is not appropriate for use in CAD tools. In the next section we discuss two alternative representations of logic functions, which are suitable for use in CAD algorithms.

8.3 Alternative Representations of Logic Functions

So far in this book, we have encountered four different forms for representing logic functions: truth tables, algebraic expressions, Venn

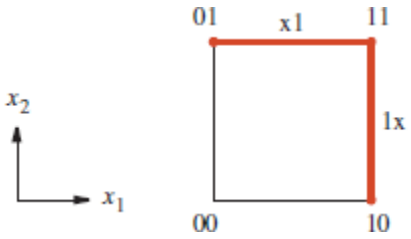
diagrams, and Karnaugh maps. Two other possibilities are introduced in this section, called cubical notation and binary decision diagrams.

8.3.1 CUBICAL REPRESENTATION

Cubical notation represents an *n*-variable logic function by mapping it onto an *n*-dimensional cube. We can illustrate this representation by first introducing some simple examples of cubes.

Two-Dimensional Cube

A two-dimensional cube is shown in Figure 8.16. The four corners in the cube are called *vertices*, which correspond to the four rows of a truth table. Each vertex is identified by two coordinates. The horizontal coordinate is assumed to correspond to variable x_1, and vertical coordinate to x_2. Thus vertex 00 is the bottom-left corner, which corresponds to row 0 in the truth table. Vertex 01 is the top-left corner, where $x_1 = 0$ and $x_2 = 1$, which corresponds to row 1 in the truth table, and so on for the other two vertices.



x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

Figure 8.16 Representation of $f(x_1,x_2) = \sum m(1,2,3)$.

We will map a function onto the cube by indicating with red circles those vertices for which $f = 1$. In Figure 8.16 $f = 1$ for vertices 01, 10, and 11. We can express the function as a set of vertices, using the notation $f = \{01, 10, 11\}$. The function *f* is also shown in the form of a truth table in the figure.

An edge joins two vertices for which the labels differ in the value of only one variable. Therefore, if two vertices for which $f = 1$ are joined by an edge, then this edge represents that portion of the function just as well as the two individual vertices. For example, $f = 1$ for vertices 10 and 11. They are joined by the edge that is labeled 1x. It is customary to use the letter x to denote the fact that the corresponding variable can be either 0 or 1. Hence 1x means that $x_1 = 1$, while x_2 can be either 0 or 1. Similarly, vertices 01 and 11 are joined by the edge labeled x1, indicating that x_1 can be either 0 or 1,

but x_1. The reader must not confuse the use of the letter x for this purpose, in contrast to the subscripted use where x_1 and x_2 refer to the variables.

Two vertices being represented by a single edge is the embodiment of the combining property 14*a* from Section 2.5. The edge 1x is the logical OR of vertices 10 and 11. It essentially defines the term x_1, which is the sum of minterms $x_1\bar{x}_2$ and x_1x_2. The property 14*a* indicates that

$$x_1\bar{x}_2 + x_1x_2 = x_1$$

Therefore, finding edges for which $f = 1$ is equivalent to applying the combining property. Of course, this is also analogous to finding pairs of adjacent cells in a Karnaugh map for which $f = 1$.

The edges 1x and x1 define fully the function in Figure 8.16; hence we can represent the function as $f = \{1x, x1\}$. This corresponds to the logic expression

$$f = x_1 + x_2$$

which is also obvious from the truth table in the figure.

Three-Dimensional Cube

Figure 8.17 illustrates a three-dimensional cube. The x_1, x_2, and x_3 coordinates are as shown on the left. Each vertex is identified by a specific valuation of the three variables. The function *f* mapped onto the cube is the function from Figure 2.48, which was used in Figure 2.52*b*. There are five vertices for which $f = 1$, namely, 000, 010, 100, 101, and 110. These vertices are joined by the five edges shown in red, namely, x00, 0x0, x10, 1x0, and 10x. Because the vertices 000, 010, 100, and 110 include all valuations of x_1 and x_2, when x_3 is 0, they can be specified by the term xx0. This term means that $f = 1$ if $x_3 = 0$, regardless of the values of x_1 and x_2. Notice that xx0 represents the front side of the cube, which is shaded in red.

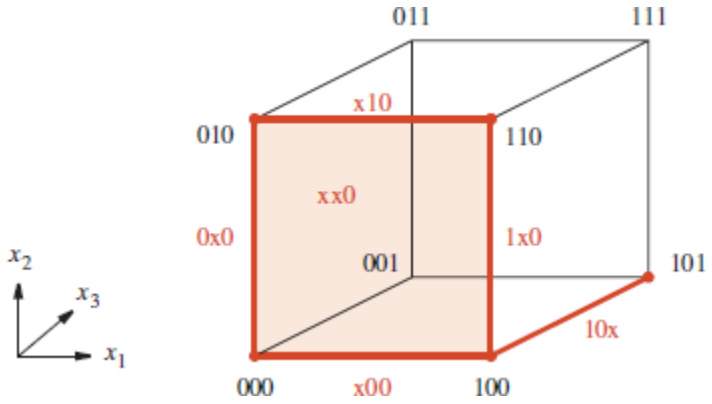


Figure 8.17 Representation of $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

From the preceding discussion it is evident that the function f can be represented in several ways. Some of the possibilities are

$$\begin{aligned} f &= \{000, 010, 100, 101, 110\} \\ &= \{0x0, 1x0, 101\} \\ &= \{x00, x10, 101\} \\ &= \{x00, x10, 10x\} \\ &= \{xx0, 10x\} \end{aligned}$$

In a physical realization each of the above terms is a product term implemented by an AND gate. Obviously, the minimum-cost circuit is obtained if $f = \{xx0, 10x\}$, which is equivalent to the logic expression

$$f = \bar{x}_3 + x_1\bar{x}_2$$

This is the expression that we derived using the Karnaugh map in Figure 2.52*b*.

Four-Dimensional Cube

Graphical images of two- and three-dimensional cubes are easy to draw. A four-dimensional cube is more difficult. It consists of 2 three-dimensional cubes with their corners connected. A simple way to visualize a four-dimensional cube is to have one cube placed inside the other cube, as depicted in Figure 8.18. We have assumed that the x_1, x_2, and x_3 coordinates are the same as in Figure 8.17, while $x_4 = 0$ defines the outer cube and $x_4 = 1$ defines the inner cube. Figure 8.18 indicates how the function f_3 of Figure

2.54 is mapped onto the four-dimensional cube. To avoid cluttering the figure with too many labels, we have labeled only those vertices for which $f_3 = 1$. Again, all edges that connect these vertices are highlighted in red.

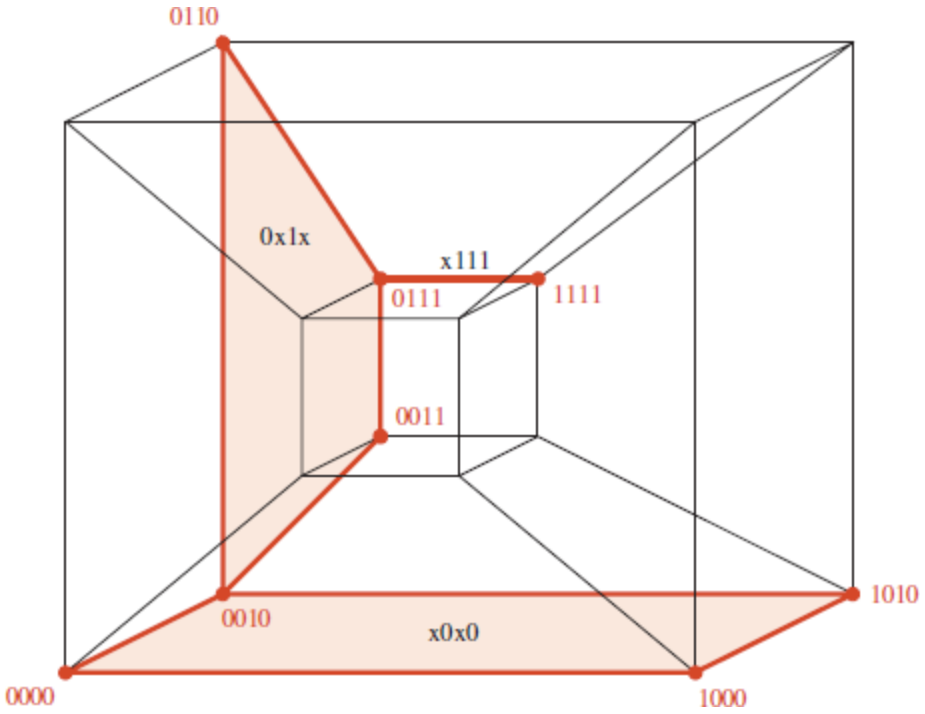


Figure 8.18 Representation of function f_3 from Figure 2.54.

There are two groups of four adjacent vertices for which $f_3 = 1$ that can be represented as planes. The group comprising 0000, 0010, 1000, and 1010 is represented by x0x0. The group 0010, 0011, 0110, and 0111 is represented by 0x1x. These planes are shaded in the figure. The function f_3 can be represented in several ways, for example

$$\begin{aligned} f_3 &= \{0000, 0010, 0011, 0110, 0111, 1000, 1010, 1111\} \\ &= \{00x0, 10x0, 0x10, 0x11, x111\} \\ &= \{x0x0, 0x1x, x111\} \end{aligned}$$

Since each x indicates that the corresponding variable can be ignored, because it can be either 0 or 1, the simplest circuit is obtained if $f = \{x0x0, 0x1x, x111\}$, which is equivalent to the expression

$$f_3 = \bar{x}_2\bar{x}_4 + \bar{x}_1x_3 + x_2x_3x_4$$

We derived the same expression in Figure 2.54.

n-Dimensional Cube

A function that has *n* variables can be mapped onto an *n*-dimensional cube. Although it is impractical to draw graphical images of cubes that have more than four variables, it is not difficult to extend the ideas introduced above to a general *n*-variable case. Because visual interpretation is not possible and because we normally use the word *cube* only for a three-dimensional structure, many people use the word *hypercube* to refer to structures with more than three dimensions. We will continue to use the word *cube* in our discussion.

It is convenient to refer to a cube as being of a certain *size* that reflects the number of vertices in the cube. Vertices have the smallest size. Each variable has a value of 0 or 1 in a vertex. A cube that has an x in one variable position is larger because it consists of two vertices. For example, the cube 1x01 consists of vertices 1001 and 1101. A cube that has two x's consists of four vertices, and so on. A cube that has *k* x's consists of 2^k vertices.

An *n*-dimensional cube has 2^n vertices. Two vertices are adjacent if they differ in the value of only one coordinate. Because there are *n* coordinates (axes in the *n*-dimensional cube), each vertex is adjacent to *n* other vertices. The *n*-dimensional cube contains cubes of lower dimensionality. Cubes of the lowest dimension are vertices. Because their dimension is zero, we will call them 0-*cubes*. Edges are cubes of dimension 1; hence we will call them 1-*cubes*. A side of a three-dimensional cube is a 2-*cube*. An entire three-dimensional cube is a 3-*cube*, and so on. In general, we will refer to a set of 2^k adjacent vertices as a *k*-*cube*.

From the examples in Figures 8.17 and 8.18, it is apparent that the largest possible *k*-*cubes* that exist for a given function are equivalent to its prime implicants. In Section 8.4 we will discuss minimization techniques that use the cubical representation of functions.

8.3.2 BINARY DECISION DIAGRAMS

Binary decision diagrams (BDDs) represent logic functions in the form of a graph. BDDs are attractive for use in CAD tools because a logic function can be represented as a BDD without requiring a lot of memory space. To

illustrate how we can derive a BDD for a function *f*, consider the two-variable truth table shown in Figure 8.19*a*. A graph that corresponds to this truth table is given in part (*b*) of the figure in the form of a *decision tree*. The circular nodes in this tree represent the input variables x_1 and x_2. The node x_1 is the root of the tree. Each node has two associated edges. For a node x_i the edge labeled 0 represents $x_i = 0$, and the edge labeled 1 corresponds to $x_i = 1$. The square nodes are the *terminal* nodes of the tree. Each terminal node specifies whether $f = 0$ or $f = 1$. Each path from the root node to a terminal node corresponds to a row in the truth table. For example, the top row of the truth table in Figure 8.19*a* has $x_1 = x_2 = 0$ and $f = 1$. In the decision tree the corresponding path starts at the node x_1, follows the edge labeled 0 to the subsequent node x_2, and then follows the edge labeled 0 to the terminal node 1.

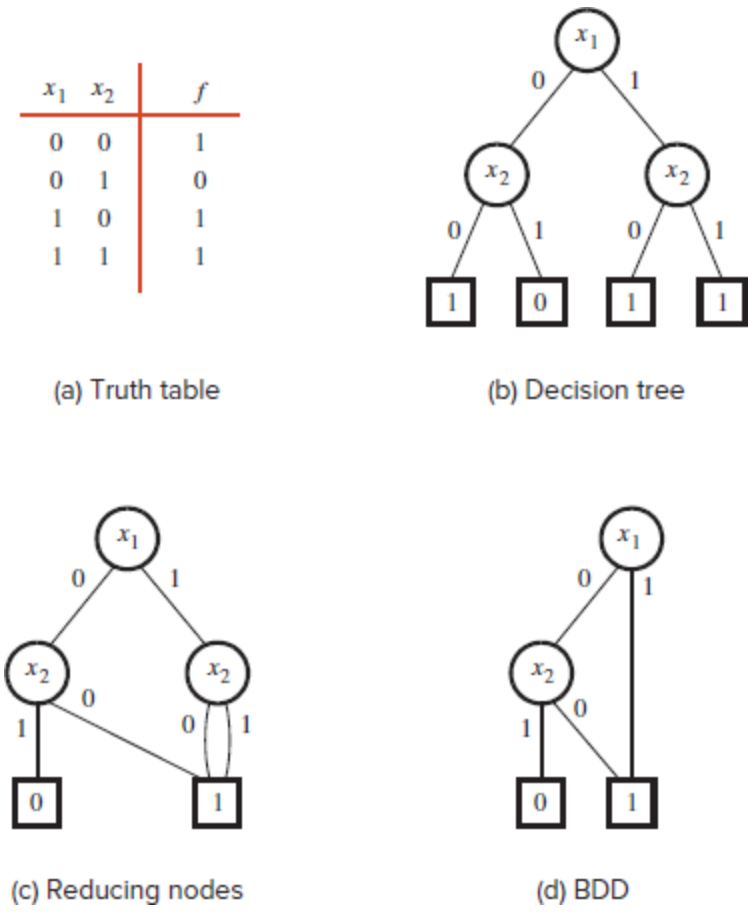


Figure 8.19 Derivation of a binary decision diagram (BDD).

The decision tree is not an efficient way of representing a logic function, because the size of the tree doubles with each input variable. But we can reduce the number of nodes in the tree as illustrated in Figure 8.19*c*. Here, we have merged identical terminal nodes such that there is only one terminal node representing each of the values 0 and 1. We can then recognize that the node x_2 on the right-hand side of the tree is redundant and can be removed, because both of its edges lead to the same subsequent node. After all identical nodes and redundant nodes have been removed from the decision tree, the resulting tree is referred to in the technical literature as a *binary decision diagram* (BDD). A BDD for the truth table in Figure 8.19*a* is given in Figure 8.19*d*. It provides a compact representation of the logic function, and can also be used to provide a low-cost implementation. In Figure 8.19*d* we can see that f has the value 1 if $x_1 = 1$ or if $x_1 = 0$ and $x_2 = 0$. Hence, we can write $f = x_1 + \bar{x}_1\bar{x}_2 = x_1 + \bar{x}_2$. In general, a BDD can be used to provide various implementations of a function, which may be in either two-level or multilevel form.

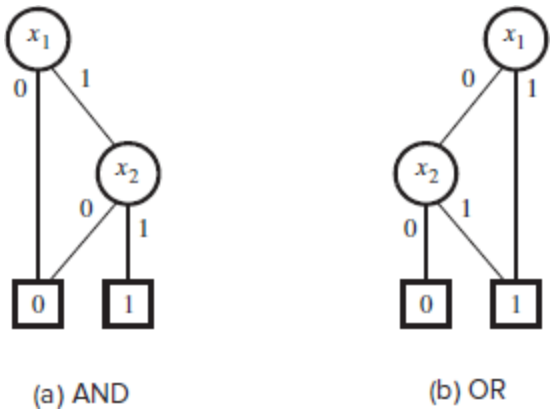


Figure 8.20 BDDs for the AND and OR functions.

Examples of BDDs for the logical AND and OR functions are given in Figure 8.20. Part (*a*) of the figure represents the function $f = x_1x_2$ and part (*b*) represents $f = x_1 + x_2$. It is easy to extend these BDDs to represent larger AND and OR functions. Figure 8.21 shows how we can derive BDDs for XOR functions. Part (*a*) gives a decision tree for the function $f = x_1 \oplus x_2$. When identical terminal nodes in the decision tree are merged, no redundant nodes are created. The resulting BDD is shown in part (*b*) of the figure. Part (*c*) shows how this BDD can be extended to represent the three-input XOR function $f = x_1 \oplus x_2 \oplus x_3$. Note that the terminal node 1 is reached from the root

node x_1 by following paths on which the total number of edges labeled 1 is odd, as required for the XOR function.

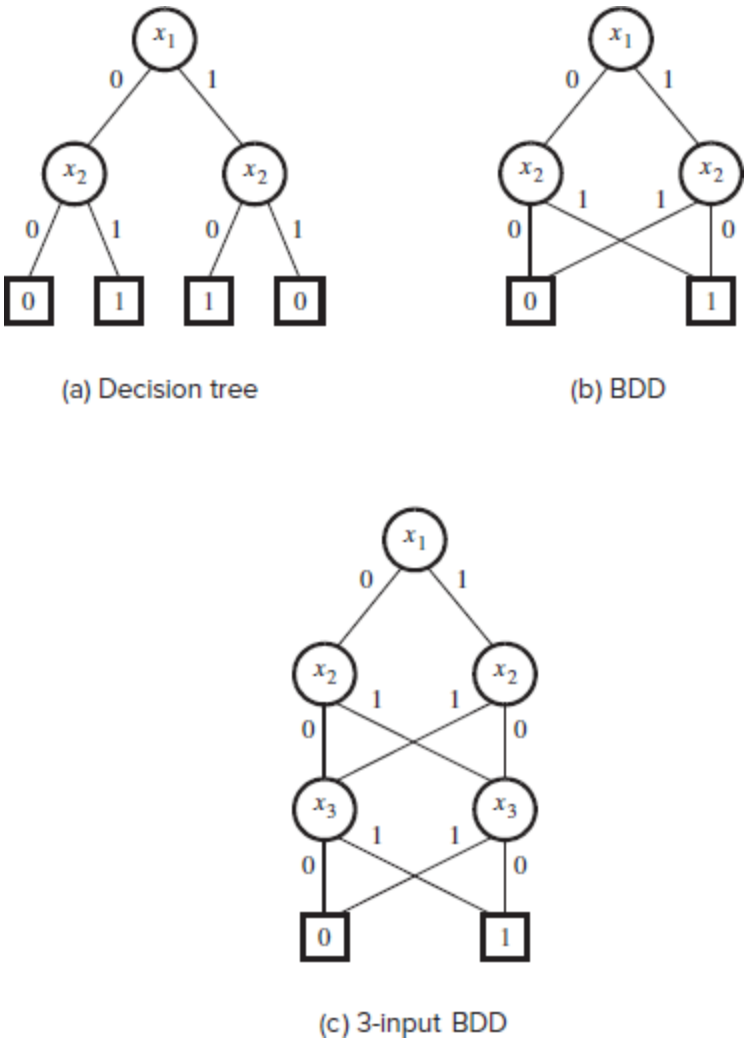


Figure 8.21 Derivation of BDDs for XOR functions.

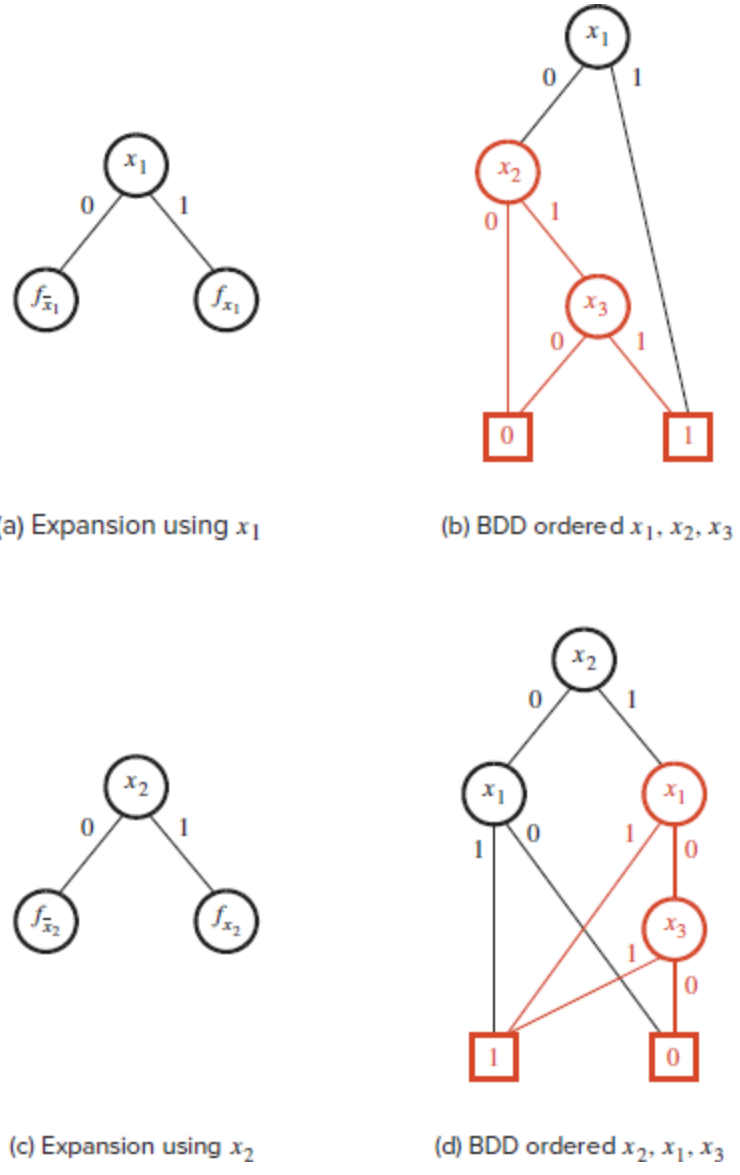
Derivation of BDDs using Shannon's Expansion

As shown in Figures 8.19 and 8.21, it is possible to derive a BDD by removing redundant nodes and merging identical nodes in a decision tree. Although this method can be used for any logic function, it is tedious when the function has more than a few input variables. Another way of deriving a BDD is to make use of Shannon's expansion theorem, which we introduced in Section 4.1.2. Recall that Shannon's expansion states that any Boolean function $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ can be written in the form

$$f(x_1, x_2, \ldots, x_i, \ldots, x_n) = \bar{x}_i f_{\bar{x}_i} + x_i f_{x_i}$$

where $f_{\bar{x}_i} = f(x_1, x_2, \ldots, 0, \ldots, x_n)$ and $f_{x_i} = f(x_1, x_2, \ldots, 1, \ldots, x_n)$.

Consider the function $f = x_1 + x_2 x_3$. Shannon's expansion with respect to the input x_1 gives $f = \bar{x}_1 f_{\bar{x}_1} + x_1 f_{x_1}$, which can be represented using the graph in Figure 8.22*a*. The cofactors $f_{\bar{x}_1}$ and f_{x_1} can be evaluated as follows:



(a) Expansion using x_1

(b) BDD ordered x_1, x_2, x_3

(c) Expansion using x_2

(d) BDD ordered x_2, x_1, x_3

Figure 8.22 Derivation of BDDs for $f = x_1 + x_2 x_3$.

$$f_{\bar{x}_1} = 0 + x_2 x_3 = x_2 x_3$$
$$f_{x_1} = 1 + x_2 x_3 = 1$$

We can now derive our BDD as shown in Figure 8.22*b*. The subfunction $f_{\bar{x}_1} = x_2 x_3$, which is highlighted in a red color, uses the BDD for the two-input AND function that we showed in Figure 8.20*a*.

A useful property of BDDs is that for a given order of the input variables there is only one possible BDD. Using the terminology from Chapter 2, this means that for a given order of the inputs a BDD is a canonical representation of the function. However, we should consider the effect on a BDD if we change the order of variables. Figure 8.22*c* corresponds to the Shannon's expansion of *f* using the order x_2, x_1, x_3. Now, node x_2 is the root of the tree and its edges point to the cofactors of *f* with respect to x_2. We have

$$f_{\bar{x}_2} = x_1 + 0 \cdot x_3 = x_1$$
$$f_{x_2} = x_1 + 1 \cdot x_3 = x_1 + x_3$$

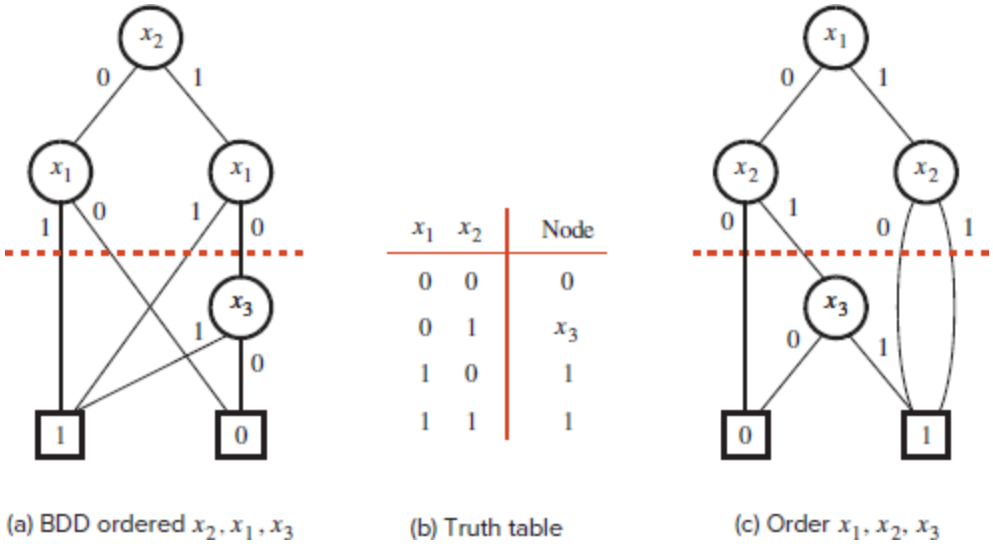
These expressions lead to the BDD in Figure 8.22*d*. The edge labeled 0 from node x_2 points to the subfunction x_1. The edge labeled 1 from node x_2 points to the subfunction $x_1 + x_3$, which is highlighted in red. This subfunction uses the same BDD that we showed for the two-input OR function in Figure 8.20*b*.

Comparing the BDDs in parts (*b*) and (*d*) of Figure 8.22, we see that the complexity of a BDD for a given function can be affected by the order of variables. In Figure 8.22*b* there are five nodes in the BDD, while in Figure 8.22*d* there are six nodes. CAD tools that implement BDDs usually employ heuristic techniques which attempt to find a good order of the input variables when deriving BDDs.

Example 8.10

Assume that we are given the BDD in Figure 8.22*d*, and that we wish to determine the effect on this BDD of swapping the positions of nodes x_1 and x_2, so that x_1 becomes the root of the BDD. The swapping operation can be performed as indicated in Figure 8.23. The first step is to isolate the nodes x_1 and x_2 from the rest of the BDD, as shown by the dotted red line in Figure 8.23*a*. Then, we enumerate the valuations of the inputs x_1 and x_2 on each path that crosses this boundary, and determine which successor node is reached. The truth table in Figure 8.23*b* provides the required information. It shows that the path with $x_1 x_2 = 00$ leads to the terminal node 0, the path

with $x_1x_2 = 01$ leads to node x_3, and the remaining paths lead to the terminal node 1. We can now create the diagram shown in part (*c*) of the figure, ensuring that all combinations of x_1x_2 lead to the correct successor nodes. We now remove the redundant node x_2 on the right-hand side of the diagram, leading to the BDD in Figure 8.22*b*.



(a) BDD ordered x_2, x_1, x_3 (b) Truth table (c) Order x_1, x_2, x_3

Figure 8.23 Reordering the nodes in a BDD.

Example 8.11

Consider the SOP expression $f = x_1x_3 + x_1x_4 + x_2x_4 + x_2x_3 + \bar{x}_1\bar{x}_2x_3x_4$. We wish to create a BDD for this function with the input order x_1, x_2, x_3, x_4. The first step is to use Shannon's expansion with respect to input x_1, which is $f = \bar{x}_1 f_{\bar{x}_1} + x_1 f_{x_1}$. The cofactors are

$$\begin{aligned} f_{\bar{x}_1} &= 0 \cdot x_3 + 0 \cdot x_4 + x_2x_4 + x_2x_3 + 1 \cdot \bar{x}_2x_3x_4 \\ &= x_2x_4 + x_2x_3 + \bar{x}_2x_3x_4 \\ f_{x_1} &= 1 \cdot x_3 + 1 \cdot x_4 + x_2x_4 + x_2x_3 + 0 \cdot \bar{x}_2x_3x_4 \\ &= x_3 + x_4 + x_2x_4 + x_2x_3 = x_3 + x_4 \end{aligned}$$

For the cofactor $f_{x_1} = x_3 + x_4$ we can use the BDD from Figure 8.20*b*. But for the cofactor $f_{\bar{x}_1} = x_2x_4 + x_2x_3 + \bar{x}_2x_3x_4$ it is useful to apply Shannon's expansion again, with respect to input x_2. This gives

$$f_{\bar{x}_1\bar{x}_2} = 0 \cdot x_4 + 0 \cdot x_3 + 1 \cdot x_3x_4 = x_3x_4$$
$$f_{\bar{x}_1x_2} = 1 \cdot x_4 + 1 \cdot x_3 + 0 \cdot x_3x_4 = x_3 + x_4$$

Now, we can use the BDDs in parts (*a*) and (*b*) of Figure 8.20 for the cofactors $f_{\bar{x}_1\bar{x}_2}$ and $f_{\bar{x}_1x_2}$, respectively. The above steps lead to the diagram in Figure 8.24*a*, where the BDD for the subfunction $x_3 + x_4$ is highlighted in red. This subfunction is used for both the cofactor f_{x_1}, which is reached when $x_1 = 1$, and the cofactor $f_{\bar{x}_1x_2}$, which is reached when $x_1 = 0$ and $x_2 = 1$. The BDD for the subfunction x_3x_4 is shown on the left-hand side of Figure 8.24*a*. This subfunction is used for the cofactor $f_{\bar{x}_1\bar{x}_2}$ and is reached when $x_1 = x_2 = 0$.

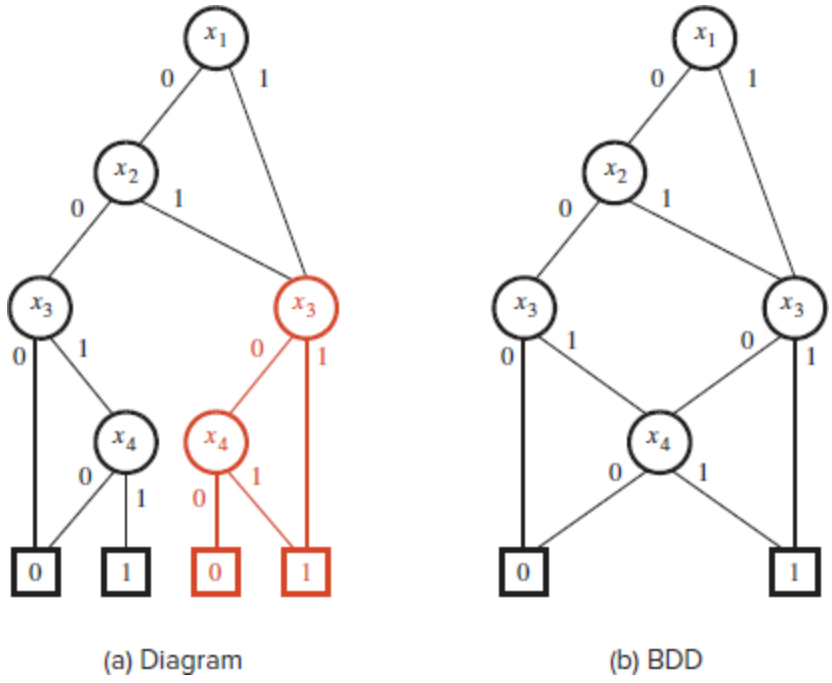


Figure 8.24 Derivation of a BDD for the function in Example 8.11.

Merging the identical terminal nodes in the graph in Figure 8.24*a*, and also merging the two identical copies of node x_4, lead to the BDD shown in Figure 8.24*b*. Since this BDD specifies that f is defined by the subfunction $x_3 + x_4$ when $x_1 = 1$ or $x_2 = 1$, and f is defined by the subfunction x_3x_4 when $x_1 = x_2 = 0$, then we can write the multilevel expression

$$f = (x_1 + x_2) \cdot (x_3 + x_4) + \bar{x}_1\bar{x}_2(x_3x_4)$$

It is easy to see how this expression relates to the SOP form above.

Practical use of BDDs

We have presented only a few introductory concepts related to BDDs, in an attempt to give the reader an appreciation for this method of representing logic functions. More detailed information can be found in various forms of literature. The BDD concept has been implemented in a number of software programs that are available through the internet. These programs are often referred to as *BDD packages*. They provide many useful functions, including the ability to create BDDs for logic functions, reorder the inputs to optimize these BDDs, and perform logical operations using BDDs. Although a detailed discussion of these BDD packages is beyond the scope of this book, the user is encouraged to search for available packages and experiment with the provided functionality. Two popular packages that are often included as part of CAD tools for logic synthesis are called BuDDy and CUDD.

8.4 Optimization Techniques Based on Cubical Representation

In this section we introduce optimization techniques that utilize the cubical representation for logic functions. Cubical representation is well suited for implementation of minimization algorithms that can be programmed and run efficiently on computers. Such algorithms are included in modern CAD tools. While the CAD tools can be used effectively without detailed knowledge of how their minimization algorithms are implemented, the reader may find it interesting to gain some insight into how this may be accomplished.

8.4.1 A Tabular Method for Minimization

A tabular approach for minimization was proposed in the 1950s by Willard Quine and Edward McCluskey. It became popular under the name *Quine-McCluskey method*. While it is not efficient enough to be used in modern CAD tools, it is a simple method that illustrates the key issues. We will present it using the cubical notation discussed in Section 8.3.

Generation of Prime Implicants

As mentioned in Section 8.3, the prime implicants of a given logic function *f* are the largest possible *k*-cubes for which $f = 1$. For incompletely specified functions, which include a set of don't-care vertices, the prime implicants are the largest *k*-cubes for which either $f = 1$ or *f* is unspecified.

Assume that the initial specification of *f* is given in terms of minterms for which $f = 1$. Also, let the don't-cares be specified as minterms. This allows us to create a list of vertices for which either $f = 1$ or it is a don't-care condition. We can compare these vertices in pairwise fashion to see if they can be combined into larger cubes. Then we can attempt to combine these new cubes into still larger cubes and continue the process until we find the prime implicants.

The basis of the method is the combining property of Boolean algebra

$$x_i x_j + x_i \bar{x}_j = x_i$$

If we have two cubes that are identical in all variables (coordinates) except one, for which one cube has the value 0 and the other has 1, then these cubes can be combined into a larger cube. For example, consider $f(x_1, \ldots, x_4) = \{1000, 1001, 1010, 1011\}$. The cubes 1000 and 1001 differ only in variable x_4; they can be combined into a new cube 100x. Similarly, 1010 and 1011 can be combined into 101x. Then we can combine 100x and 101x into a larger cube 10xx, which means that the function can be expressed simply as $f = x_1 \bar{x}_2$.

Figure 8.25 shows how we can generate the prime implicants of the function, *f*, in Figure 2.58. The function is defined as

List 1

0	0 0 0 0	✓
4	0 1 0 0	✓
8	1 0 0 0	✓
10	1 0 1 0	✓
12	1 1 0 0	✓
11	1 0 1 1	✓
13	1 1 0 1	✓
15	1 1 1 1	✓

List 2

0,4	0 x 0 0	✓
0,8	x 0 0 0	✓
8,10	1 0 x 0	
4,12	x 1 0 0	✓
8,12	1 x 0 0	✓
10,11	1 0 1 x	
12,13	1 1 0 x	
11,15	1 x 1 1	
13,15	1 1 x 1	

List 3

0,4,8,12	x x 0 0

Figure 8.25 Generation of prime implicants for the function in Figure 2.58.

$$f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 10, 11, 12, 13, 15)$$

There are no don't-care conditions. Since larger cubes can be generated only from the minterms that differ in just one variable, we can reduce the number of pairwise comparisons by placing the minterms into groups such that the cubes in each group have the same number of 1s, and sort the groups by the number of 1s. Thus, it will be necessary to compare each cube in a given group only with all cubes in the immediately preceding group. In Figure 8.25, the minterms are ordered in this way in list 1. (Note that we indicated the decimal equivalents of the minterms as well, to facilitate our discussion.) The minterms, which are also called 0-cubes as explained in Section 8.3, can be combined into 1-cubes shown in list 2. To make the entries easily understood we indicated the minterms that are combined to form each 1-cube. Next, we check if the 0-cubes are included in the 1-cubes and place a check mark beside each cube that is included. We now generate 2-cubes from the 1-cubes in list 2. The only 2-cube that can be generated is xx00, which is placed in list 3. Again, the check marks are placed against the 1-cubes that are included in the 2-cube. Since there exists just one 2-cube, there can be no 3-cubes for this function. The cubes in each list without a check mark are the prime implicants of *f*. Therefore, the set, *P*, of prime implicants is

$$\begin{aligned} P &= \{10x0, 101x, 110x, 1x11, 11x1, xx00\} \\ &= \{p_1, p_2, p_3, p_4, p_5, p_6\} \end{aligned}$$

Determination of a Minimum Cover

Having generated the set of all prime implicants, it is necessary to choose a minimum-cost subset that covers all minterms for which $f = 1$. As a simple measure we will assume that the cost is directly proportional to the number of inputs to all gates, which means to the number of literals in the prime implicants chosen to implement the function.

To find a minimum-cost cover, we construct a *prime implicant cover table* in which there is a row for each prime implicant and a column for each minterm that must be covered. Then we place check marks to indicate the minterms covered by each prime implicant. Figure 8.26*a* shows the table for the prime implicants derived in Figure 8.25. If there is a single check mark in some column of the cover table, then the prime implicant

that covers the minterm of this column is *essential* and it must be included in the final cover. Such is the case with P_6, which is the only prime implicant that covers minterms 0 and 4. The next step is to remove the row(s) corresponding to the essential prime implicants and the column(s) covered by them. Hence we remove P_6 and columns 0, 4, 8, and 12, which leads to the table in Figure 8.26*b*.

Prime implicant	Minterm 0	4	8	10	11	12	13	15
P_1 = 1 0 x 0			✓	✓				
P_2 = 1 0 1 x				✓	✓			
P_3 = 1 1 0 x						✓	✓	
P_4 = 1 x 1 1					✓			✓
P_5 = 1 1 x 1							✓	✓
P_6 = x x 0 0	✓	✓	✓			✓		

(a) Initial prime implicant cover table

Prime implicant	Minterm 10	11	13	15
P_1	✓			
P_2	✓	✓		
P_3			✓	
P_4		✓		✓
P_5			✓	✓

(b) After the removal of essential prime implicants

Prime implicant	Minterm 10	11	13	15
P_2	✓	✓		
P_4		✓		✓
P_5			✓	✓

(c) After the removal of dominated rows

Figure 8.26 Selection of a cover for the function in Figure 2.58.

Now, we can use the concept of *row dominance* to reduce the cover table. Observe that P_1 covers only minterm 10 while P_2 covers both 10 and 11. We

say that p_2 *dominates* p_1. Since the cost of p_2 is the same as the cost of p_1, it is prudent to choose p_2 rather than p_1, so we will remove p_1 from the table. Similarly, p_5 dominates p_3, hence we will remove p_3 from the table. Thus, we obtain the table in Figure 8.26*c*. This table indicates that we must choose p_2 to cover minterm 10 and p_5 to cover minterm 13, which also takes care of covering minterms 11 and 15. Therefore, the final cover is

$$\begin{aligned} C &= \{p_2, p_5, p_6\} \\ &= \{101\text{x}, 11\text{x}1, \text{xx}00\} \end{aligned}$$

which means that the minimum-cost implementation of the function is

$$f = x_1\bar{x}_2x_3 + x_1x_2x_4 + \bar{x}_3\bar{x}_4$$

This is the same expression as the one derived from Figure 2.58.

In this example we used the concept of row dominance to reduce the cover table. We removed the dominated rows because they cover fewer minterms and the cost of their prime implicants is the same as the cost of the prime implicants of the dominating rows. However, a dominated row should not be removed if the cost of its prime implicant is less than the cost of the dominating row's prime implicant.

The tabular method can be used with don't-care conditions as illustrated in the following example.

Example 8.12

The don't-care minterms are included in the initial list in the same way as the minterms for which $f = 1$. Consider the function

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 5, 6, 7, 8, 9, 13) + D(1, 12, 15)$$

We encourage the reader to derive a Karnaugh map for this function as an aid in visualizing the derivation that follows. Figure 8.27 depicts the generation of prime implicants, producing the result

$$\begin{aligned} P &= \{00\text{x}0, 0\text{x}10, 011\text{x}, \text{x}00\text{x}, \text{xx}01, 1\text{x}0\text{x}, \text{x}1\text{x}1\} \\ &= \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\} \end{aligned}$$

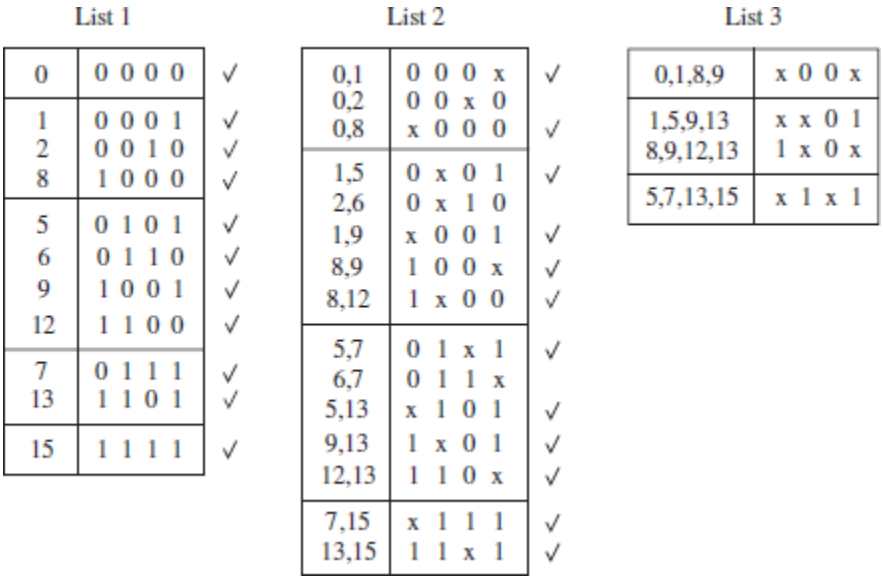


Figure 8.27 Generation of prime implicants for the function in Example 8.12.

The initial prime implicant cover table is shown in Figure 8.28*a*. The don't-care minterms are not included in the table because they do not have to be covered. There are no essential prime implicants. Examining this table, we see that column 8 has check marks in the same rows as column 9. Moreover, column 9 has an additional check mark in row p_5. Hence column 9 dominates column 8. We refer to this as the concept of *column dominance*. When one column dominates another, we can remove the dominating column, which is column 9 in this case. Note that this is in contrast to rows where we remove dominated (rather than dominating) rows. The reason is that when we choose a prime implicant to cover the minterm that corresponds to the dominated column, this prime implicant will also cover the minterm corresponding to the dominating column. In our example, choosing either p_4 or p_6 covers both minterms 8 and 9. Similarly, column 13 dominates column 5, hence column 13 can be deleted.

Prime implicant	Minterm 0	2	5	6	7	8	9	13
$p_1 = 0\ 0\ x\ 0$	✓	✓						
$p_2 = 0\ x\ 1\ 0$		✓		✓				
$p_3 = 0\ 1\ 1\ x$				✓	✓			
$p_4 = x\ 0\ 0\ x$	✓					✓	✓	
$p_5 = x\ x\ 0\ 1$			✓				✓	✓
$p_6 = 1\ x\ 0\ x$						✓	✓	✓
$p_7 = x\ 1\ x\ 1$			✓		✓			✓

(a) Initial prime implicant cover table

Prime implicant	Minterm 0	2	5	6	7	8
$p_1 = 0\ 0\ x\ 0$	✓	✓				
$p_2 = 0\ x\ 1\ 0$		✓		✓		
$p_3 = 0\ 1\ 1\ x$				✓	✓	
$p_4 = x\ 0\ 0\ x$	✓					✓
$p_5 = x\ x\ 0\ 1$			✓			
$p_6 = 1\ x\ 0\ x$						✓
$p_7 = x\ 1\ x\ 1$			✓		✓	

(b) After the removal of columns 9 and 13

Prime implicant	Minterm 0	2	5	6	7	8
p_1	✓	✓				
p_2		✓		✓		
p_3				✓	✓	
p_4	✓					✓
p_7			✓		✓	

(c) After the removal of rows p_5 and p_6

Prime implicant	Minterm 2	6
p_1	✓	
p_2	✓	✓
p_3		✓

(d) After including p_4 and p_7 in the cover

Figure 8.28 Selection of a cover for the function in Example 8.12.

After removing columns 9 and 13, we obtain the reduced table in Figure 8.28*b*. In this table row p_4 dominates p_6 and row p_7 dominates p_5. This means that p_5 and p_6 can be removed, giving the table in Figure 8.28*c*. Now, p_4 and p_7 are essential to cover minterms 8 and 5, respectively. Thus, the table in

Figure 8.28*d* is obtained, from which it is obvious that p_2 covers the remaining minterms 2 and 6. Note that row p_2 dominates both rows p_1 and p_3.

The final cover is

$$\begin{aligned} C &= \{p_2, p_4, p_7\} \\ &= \{0x10, x00x, x1x1\} \end{aligned}$$

and the function is implemented as

$$f = \bar{x}_1 x_3 \bar{x}_4 + \bar{x}_2 \bar{x}_3 + x_2 x_4$$

In Figures 8.26 and 8.28, we used the concept of row and column dominance to reduce the cover table. This is not always possible, as illustrated in the following example.

Example 8.13

Consider the function

$$f(x_1, \ldots, x_4) = \sum m(0, 3, 10, 15) + D(1, 2, 7, 8, 11, 14)$$

The prime implicants for this function are

$$\begin{aligned} P &= \{00xx, x0x0, x01x, xx11, 1x1x\} \\ &= \{p_1, p_2, p_3, p_4, p_5\} \end{aligned}$$

The initial prime implicant cover table is shown in Figure 8.29*a*. There are no essential prime implicants. Also, there are no dominant rows or columns. Moreover, all prime implicants have the same cost because each of them is implemented with two literals. Thus, the table does not provide any clues that can be used to select a minimum-cost cover.

Prime implicant	Minterm 0	3	10	15
$p_1 = 0\ 0\ x\ x$	✓	✓		
$p_2 = x\ 0\ x\ 0$	✓		✓	
$p_3 = x\ 0\ 1\ x$		✓	✓	
$p_4 = x\ x\ 1\ 1$		✓		✓
$p_5 = 1\ x\ 1\ x$			✓	✓

(a) Initial prime implicant cover table

Prime implicant	Minterm 0	15
p_1	✓	
p_2	✓	
p_4		✓
p_5		✓

(b) After including p_3 in the cover

Prime implicant	Minterm 0	3	10	15
p_1	✓	✓		
p_2	✓		✓	
p_4		✓		✓
p_5			✓	✓

(c) After excluding p_3 from the cover

Figure 8.29 Selection of a cover for the function in Example 8.13.

A good practical approach is to use the concept of *branching*. We can choose any prime implicant, say p_3, and first choose to include this prime implicant in the final cover. Then we can determine the rest of the final cover in the usual way and compute its cost. Next we try the other possibility by excluding p_3 from the final cover and determine the resulting cost. We compare the costs and choose the less expensive alternative.

Figure 8.29*b* gives the cover table that is left if p_3 is included in the final cover. The table does not include minterms 3 and 10 because they are covered by p_3. The table indicates that a complete cover must include either

p_1 or p_2 to cover minterm 0 and either p_4 or p_5 to cover minterm 15. Therefore, a complete cover can be

$$C = \{p_1, p_3, p_4\}$$

The alternative of excluding p_3 leads to the cover table in Figure 8.29*c*. Here, we see that a minimum-cost cover requires only two prime implicants. One possibility is to choose p_1 and p_5. The other possibility is to choose p_2 and p_4. Hence a minimum-cost cover is just

$$\begin{aligned} C_{min} &= \{p_1, p_5\} \\ &= \{00xx, 1x1x\} \end{aligned}$$

The function is realized as

$$f = \bar{x}_1\bar{x}_2 + x_1x_3$$

Summary of the Tabular Method

The tabular method can be summarized as follows:

1. Starting with a list of cubes that represent the minterms where $f = 1$ or a don't-care condition, generate the prime implicants by successive pairwise comparisons of the cubes.
2. Derive a cover table which indicates the minterms where $f = 1$ that are covered by each prime implicant.
3. Include the essential prime implicants (if any) in the final cover and reduce the table by removing both these prime implicants and the covered minterms.
4. Use the concept of row and column dominance to reduce the cover table further. A dominated row is removed only if the cost of its prime implicant is greater than or equal to the cost of the dominating row's prime implicant.
5. Repeat steps 3 and 4 until the cover table is either empty or no further reduction of the table is possible.

6. If the reduced cover table is not empty, then use the branching approach to determine the remaining prime implicants that should be included in a minimum cost cover.

The tabular method illustrates how an algebraic technique can be used to generate the prime implicants. It also shows a simple approach for dealing with the covering problem, to find a minimum-cost cover. The method has some practical limitations. In practice, functions are seldom defined in the form of minterms. They are usually given either in the form of algebraic expressions or as sets of cubes. The need to start the minimization process with a list of minterms means that the expressions or sets have to be expanded into this form. This list may be very large. As larger cubes are generated, there will be numerous comparisons performed and the computation will be slow. Using the cover table to select the optimal set of prime implicants is also computationally intensive when large functions are involved.

Many algebraic techniques have been developed, which aim to reduce the time that it takes to generate the optimal covers. While most of these techniques are beyond the scope of this book, we will briefly discuss one possible approach in the next section. A reader who intends to use the CAD tools, but is not interested in the details of automated minimization, may skip this section without loss of continuity.

8.4.2 A CUBICAL TECHNIQUE FOR MINIMIZATION

Assume that the initial specification of a function f is given in terms of implicants that are not necessarily either minterms or prime implicants. Then it is convenient to define an operation that will generate other implicants that are not given explicitly in the initial specification, but which will eventually lead to the prime implicants of f. One such possibility is known as the $*$-*product* operation, which is usually pronounced the "star-product" operation. We will refer to it simply as the $*$-*operation*.

*-Operation

The *-operation provides a simple way of deriving a new cube by combining two cubes that differ in the value of only one variable. Let A and $B = B_1B_2 \cdots B_n$ be two cubes that are implicants of an n-variable function. Thus each coordinate A_i and B_i is specified as having the value 0, 1, or x. There

are two distinct steps in the $*$-operation. First, the $*$-operation is evaluated for each pair A_i and B_i, in coordinates $i = 1, 2, \ldots, n$, according to the table in Figure 8.30. Then based on the results of using the table, a set of rules is applied to determine the overall result of the $*$-operation. The table in Figure 8.30 defines the coordinate $*$-operation, $A_i * B_i$. It specifies the result of $A_i * B_i$ for each possible combination of values of A_i and B_i. This result is the intersection (i.e., the common part) of A and B in this coordinate. Note that when A_i and B_i have the opposite values (0 and 1, or vice versa), the result of the coordinate $*$-operation is indicated by the symbol o. We say that the intersection of A_i and B_i is empty. Using the table, the complete $*$-operation for A and B is defined as follows:

A_i \ B_i	0	1	x
0	0	ϕ	0
1	ϕ	1	1
x	0	1	x

$A_i * B_i$

Figure 8.30 The coordinate $*$-operation.

$C = A * B$, such that

1. $C = \emptyset$ if $A_i * B_i = \emptyset$ for more than one i.
2. Otherwise, $C_i = A_i * B_i$ when $A_i * B_i \neq \emptyset$, and $C_i = \text{x}$ for the coordinate where $A_i * B_i = \emptyset$.

For example, let $A = \{0\text{x}0\}$ and $B = \{111\}$. Then $A_1 * B_1 = 0 * 1 = \emptyset$, $A_2 * B_2 = \text{x} * 1 = 1$, and $A_3 * B_3 = 0 * 1 = \emptyset$. Because the result is $\emptyset$ in two coordinates, it follows from condition 1 that $A * B = \emptyset$. In other words, these two cubes cannot be combined into another cube, because they differ in two coordinates.

As another example, consider $A = \{11\text{x}\}$ and $B = \{10\text{x}\}$. In this case $A_1 * B_1 = 1 * 1 = 1$, $A_2 * B_2 = 1 * 0 = \emptyset$ o, and $A_3 * B_3 = \text{x}$ x $*$ x = x. According to condition 2 above, $C_1 = 1$, $C_2 =$ x, and $C_3 =$ x, which gives $C = A * B = \{1\text{xx}\}$. A larger 2-cube is created from two 1-cubes that differ in one coordinate only.

The result of the $*$-operation may be a smaller cube than the two cubes involved in the operation. Consider $A = \{1\text{x}1\}$ and $B = \{11\text{x}\}$. Then $C = A * B = \{111\}$. Notice that C is included in both A and B, which means that

this cube will not be useful in searching for prime implicants. Therefore, it should be discarded by the minimization algorithm.

As a final example, consider $A = \{x10\}$ and $B = \{0x1\}$. Then $C = A * B = \{01x\}$. All three of these cubes are the same size, but *C* is not included in either *A* or *B*. Hence *C* has to be considered in the search for prime implicants. The reader may find it helpful to draw a Karnaugh map to see how cube *C* is related to cubes *A* and *B*.

Using the *-Operation to Find Prime Implicants

The essence of the *-operation is to find new cubes from pairs of existing cubes. In particular, it is of interest to find new cubes that are not included in the existing cubes. A procedure for finding the prime implicants may be organized as follows.

Suppose that a function *f* is specified by means of a set of implicants that are represented as cubes. Let this set be denoted as the cover C^k of *f*. Let c^i and c^j be any two cubes in C^k. Then apply the $*$-operation to all pairs of cubes in C^k; let G^{k+1} be the set of newly generated cubes. Hence

$$G^{k+1} = c^i * c^j \text{ for all } c^i, c^j \epsilon\, C^k$$

Now a new cover for *f* may be formed by using the cubes in C^k and G^{k+1}. Some of these cubes may be redundant because they are included in other cubes; they should be removed. Let the new cover be

$$C^{k+1} = C^k \cup G^{k+1} - \text{redundant cubes}$$

where $\cup$ denotes the logical union of two sets, and the minus sign (–) denotes the removal of elements of a set. If $C^{k+1} \neq C^k$, then a new cover C^{k+2} is generated using the same process. If $C^{k+1} = C^k$, then the cubes in the cover are the prime implicants of *f*. For an *n*-variable function, it is necessary to repeat the step at most *n* times.

Redundant cubes that have to be removed are identified through pairwise comparison of cubes. Cube $A = A_1A_2 \cdots A_n$ should be removed if it is included in some cube $B = B_1B_2 \cdots B_n$, which is the case if $A_i = B_i$ or $B_i =$ x for every coordinate *i*.

Example 8.14

Consider the function $f(x_1, x_2, x_3)$ of Figure 2.56. Assume that f is initially specified as a set of vertices that correspond to the minterms, m_0, m_1, m_2, m_3, and m_7. Hence let the initial cover be $C^0 = \{000, 001, 010, 011, 111\}$. Using the *-operation to generate a new set of cubes, we obtain $G^1 = \{00x, 0x0, 0x1, 01x, x11\}$. Then $C^1 = C^0 \cup G^1$ – redundant cubes. Observe that each cube in C^0 is included in one of the cubes in G^1; therefore, all cubes in C^0 are redundant. Thus $C^1 = G^1$.

The next step is to apply the $*$-operation to the cubes in C^1, which yields $G^2 = \{000, 001, 0xx, 0x1, 010, 01x, 011\}$. Note that all of these cubes are included in the cube 0xx; therefore, all but 0xx are redundant. Now it is easy to see that

$$\begin{aligned} C^2 &= C^1 \cup G^2 - \text{redundant terms} \\ &= \{x11, 0xx\} \end{aligned}$$

since all cubes of C^1, except x11, are redundant because they are covered by 0xx.

Applying the *-operation to C^2 yields $G^3 = \{011\}$ and

$$\begin{aligned} C^3 &= C^2 \cup G^3 - \text{redundant terms} \\ &= \{x11, 0xx\} \end{aligned}$$

Since $C^3 = C^2$, the conclusion is that the prime implicants of f are the cubes $\{x11, 0xx\}$, which represent the product terms x_2x_3 and $\bar{x}_1$. This is the same set of prime implicants that we derived using a Karnaugh map in Figure 2.56.

Observe that the derivation of prime implicants in this example is similar to the tabular method explained in Section 8.4.1 because the starting point was a function, f, given as a set of minterms.

Example 8.15

As another example, consider the four-variable function of Figure 2.57. Assume that this function is initially specified as the cover $C^0 = \{0101, 1101, 1110, 011x, x01x\}$. Then successive applications of the *-operation and removing the redundant terms gives

$$C^1 = \{x01x, x101, 01x1, x110, 1x10, 0x1x\}$$
$$C^2 = \{x01x, x101, 01x1, 0x1x, xx10\}$$
$$C^3 = C^2$$

Therefore, the prime implicants are $\bar{x}_2x_3$, $x_2\bar{x}_3x_4$, $\bar{x}_1x_2x_4$, $\bar{x}_1x_3$, and $x_3\bar{x}_4$.

Determination of Essential Prime Implicants

From a cover that consists of all prime implicants, it is necessary to extract a minimal cover. As we saw in Section 2.12.2, all *essential* prime implicants must be included in the minimal cover. To find the essential prime implicants, it is useful to define an operation that determines a part of a cube (implicant) that is *not* covered by another cube. One such operation is called the *#-operation* (pronounced the "sharp operation''), which is defined as follows.

#-Operation

Again, let $A = A_1A_2 \cdots A_n$ and $B = B_1B_2 \cdots B_n$ be two cubes (implicants) of an n-variable function. The sharp operation $A\#B$ leaves as a result "that part of A that is not covered by B." Similar to the $*$-operation, the #-operation has two steps: $A_i\#B_i$ is evaluated for each coordinate i, and then a set of rules is applied to determine the overall result. The sharp operation for each coordinate is defined in Figure 8.31. After this operation is performed for all pairs (A_i, B_i), the complete #-operation is defined as follows:

A_i \ B_i	0	1	x
0	ϵ	ϕ	ϵ
1	ϕ	ϵ	ϵ
x	1	0	ϵ

$A_i \# B_i$

Figure 8.31 The coordinate #-operation.

$$C = A\#B, \text{ such that}$$

1. $C = A$ if $A_i\#B_i = \emptyset$ for some i.
2. $C = \emptyset$ if $A_i\#B_i = \epsilon$ for all i.

3. Otherwise, $C = \bigcup_i (A_1, A_2, \ldots, \overline{B}_i, \ldots, A_n)$ where the union is for all i for which $A_i = x$ and $B_i \neq x$.

The first condition corresponds to the case where cubes A and B do not intersect at all; namely, A and B differ in the value of at least one variable, which means that no part of A is covered by B. For example, let $A = 0x1$ and $B = 11x$. The coordinate #-products are $A_1 \# B_1 = \emptyset$, $A_2 \# B_2 = 0$, and $A_3 \# B_3 = \varepsilon$. Then from rule 1 it follows that 0x1 # 11x = 0x1. The second condition reflects the case where A is fully covered by B. For example, 0x1 # 0xx $= \emptyset$. The third condition is for the case where only a part of A is covered by B. In this case the #-operation generates one or more cubes. Specifically, it generates one cube for each coordinate i that is x in A_i, but is not x in B_i. Each cube generated is identical to A, except that A_i is replaced by $\overline{B}_i$. For example, 0xx # 01x = 00x, and 0xx # 010 ={ 00x, 0x1}.

We will now show how the #-operation can be used to find the essential prime implicants. Let P be the set of all prime implicants of a given function f. Let p^i denote one prime implicant in the set P and let DC denote the don't-care vertices for f. (We use superscripts to refer to different prime implicants in this section because we are using subscripts to refer to coordinate positions in cubes.) Then p^i is an essential prime implicant if and only if

$$p^i \# (P - p^i) \# DC \neq \emptyset$$

This means that p^i is essential if there exists at least one vertex for which $f = 1$ that is covered by p^i, but not by any other prime implicant. The #-operation is also performed with the set of don't-care cubes because vertices in p^i that correspond to don't-care conditions are not essential to cover. The meaning of $p^i \# (P - p^i)$ is that the #-operation is applied successively to each prime implicant in P. For example, consider $P = \{p^1, p^2, p^3, p^4\}$ and $DC = \{d^1, d^2\}$. To check whether p^3 is essential, we evaluate

$$((((p^3 \# p^1) \# p^2) \# p^4) \# d^1) \# d^2$$

If the result of this expression is not o, then p^3 is essential.

Example 8.16

In Example 8.14 we determined that the cubes x11 and 0xx are the prime implicants of the function f in Figure 2.56. We can discover whether each of these prime implicants is essential as follows

$$x11 \# 0xx = 111 \neq \emptyset$$
$$0xx \# x11 = \{00x, 0x0\} \neq \emptyset$$

The cube x11 is essential because it is the only prime implicant that covers the vertex 111, for which $f = 1$. The prime implicant 0xx is essential because it is the only one that covers the vertices 000, 001, and 010. This can be seen in the Karnaugh map in Figure 2.56.

Example 8.17

In Example 8.15 we found that the prime implicants of the function in Figure 2.57 are $P = \{x01x, x101, 01x1, 0x1x, xx10\}$. Because this function has no don't-cares, we compute

$$x01x \# (P - x01x) = 1011 \neq \emptyset$$

This is computed in the following steps: x01x # x101 = x01x, then x01x # 01x1 = x01x, then x01x # 0x1x = 101x, and finally 101x # xx10 = 1011. Similarly, we obtain

$$x101 \# (P - x101) = 1101 \neq \emptyset$$
$$01x1 \# (P - 01x1) = \emptyset$$
$$0x1x \# (P - 0x1x) = \emptyset$$
$$xx10 \# (P - xx10) = 1110 \neq \emptyset$$

Therefore, the essential prime implicants are x01x, x101, and xx10 because they are the only ones that cover the vertices 1011, 1101, and 1110, respectively. This is obvious from the Karnaugh map in Figure 2.57.

When checking whether a cube A is essential, the #-operation with one of the cubes in $P - A$ may generate multiple cubes. If so, then each of these cubes has to be checked using the #-operation with all of the remaining cubes in $P - A$.

Complete Procedure for Finding a Minimal Cover

Having introduced the $*$- and #-operations, we can now outline a complete procedure for finding a minimal cover for any *n*-variable function. Assume that the function *f* is specified in terms of vertices for which $f = 1$; these vertices are often referred to as the *ON-set* of the function. Also, assume that the don't-care conditions are specified as a *DC-set*. Then the initial cover for *f* is a union of the ON and DC sets.

Prime implicants of *f* can be generated using the $*$-operation, and the #-operation can be used to find the essential prime implicants. If the essential prime implicants cover the entire ON-set, then they form the minimum-cost cover for *f*. Otherwise, it is necessary to include other prime implicants until all vertices in the ON-set are covered.

A nonessential prime implicant p^i should be deleted if there exists a less-expensive prime implicant p^j that covers all vertices of the ON-set that are covered by p^i. If the remaining nonessential prime implicants have the same cost, then a possible heuristic approach is to arbitrarily select one of them, include it in the cover, and determine the rest of the cover. Then an alternative cover is generated by excluding this prime implicant, and the lower-cost cover is chosen for implementation. We already used this approach, which is often referred to as the *branching* heuristic, in Sections 2.12.2 and 8.4.1.

The preceding discussion can be summarized in the form of the following minimization procedure:

1. Let $C^0 = ON \cup DC$ be the initial cover of function *f* and its don't-care conditions.
2. Find all prime implicants of C^0 using the $*$-operation; let *P* be this set of prime implicants.
3. Find the essential prime implicants using the #-operation. A prime implicant p^i is essential if $p^i \# (P - p^i) \# DC \neq \emptyset$. If the essential prime implicants cover all vertices of the ON-set, then these implicants form the minimum-cost cover.
4. Delete any nonessential p^i that is more expensive (i.e., a smaller cube) than some other prime implicant p^j if $p^i \# DC \# p^j = \emptyset$.

5. Choose the lowest-cost prime implicants to cover the remaining vertices of the ON-set. Use the branching heuristic on the prime implicants of equal cost and retain the cover with the lowest cost.

Example 8.18

To illustrate the minimization procedure, we will use the function

$$f(x_1, x_2, x_3, x_4, x_5) = \sum m(0, 1, 4, 8, 13, 15, 20, 21, 23, 26, 31) + D(5, 10, 24, 28)$$

To help the reader follow the discussion, this function is also shown in the form of a Karnaugh map in Figure 8.32.

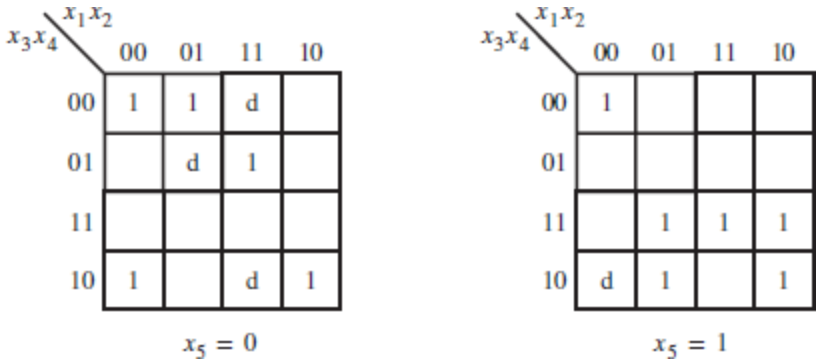


Figure 8.32 The function for Example 8.18.

Instead of f being specified in terms of minterms, let us assume that f is given as the following SOP expression

$$f = \bar{x}_1\bar{x}_3\bar{x}_4\bar{x}_5 + x_1x_2\bar{x}_3x_4\bar{x}_5 + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4x_5 + \bar{x}_1x_2x_3x_5 + x_1\bar{x}_2x_3x_5 + x_1x_3x_4x_5 + \bar{x}_2x_3\bar{x}_4\bar{x}_5$$

Also, we will assume that don't-cares are specified using the expression

$$\text{DC} = x_1x_2\bar{x}_4\bar{x}_5 + \bar{x}_1x_2\bar{x}_3x_4\bar{x}_5 + \bar{x}_1\bar{x}_2x_3\bar{x}_4x_5$$

Thus, the ON-set expressed as cubes is

$$\text{ON} = \{0\text{x}000,\ 11010,\ 00001,\ 011\text{x}1,\ 101\text{x}1,\ 1\text{x}111,\ \text{x}0100\}$$

and the don't-care set is

$$DC = \{11x00, 01010, 00101\}$$

The initial cover C^0 consists of the ON-set and the DC-set:

$$C^0 = \{0x000, 11010, 00001, 011x1, 101x1, 1x111, x0100, 11x00, 01010, 00101\}$$

Using the $*$-operation, the subsequent covers obtained are

$$C^1 = \{0x000, 011x1, 101x1, 1x111, x0100, 11x00, 0000x, 00x00, x1000, 010x0, 110x0, x1010, 00x01, x1111, 0x101, 1010x, x0101, 1x100, 0010x\}$$
$$C^2 = \{0x000, 011x1, 101x1, 1x111, 11x00, x1111, 0x101, 1x100, x010x, 00x0x, x10x0\}$$
$$C^3 = C^2$$

Therefore, $P = C^2$.

Using the #-operation, we find that there are two essential prime implicants: 00x0x (because it is the only one that covers the vertex 00001) and x10x0 (because it is the only one that covers the vertex 11010). The minterms of *f* covered by these two prime implicants are $m(0, 1, 4, 8, 26)$.

Next, we find that 1x100 can be deleted because the only ON-set vertex that it covers is 10100 (m_{20}), which is also covered by x010x and the cost of this prime implicant is lower. Note that having removed 1x100, the prime implicant x010x becomes essential because none of the other remaining prime implicants covers the vertex 10100. Therefore, x010x has to be included in the final cover. It covers $m(20, 21)$.

There remains to find prime implicants to cover $m(13, 15, 23, 31)$. Using the branching heuristic, the lowest-cost cover is obtained by including the prime implicants 011x1 and 1x111. Thus the final cover is

$$C_{minimum} = \{00x0x, x10x0, x010x, 011x1, 1x111\}$$

The corresponding sum-of-products expression is

$$f = \bar{x}_1\bar{x}_2\bar{x}_4 + x_2\bar{x}_3\bar{x}_5 + \bar{x}_2x_3\bar{x}_4 + \bar{x}_1x_2x_3x_5 + x_1x_3x_4x_5$$

Although this procedure is tedious when performed by hand, it is not difficult to write a computer program to implement the algorithm automatically. The reader should check the validity of our solution by finding the optimal realization from the Karnaugh map in Figure 8.32.

8.4.3 PRACTICAL CONSIDERATIONS

The purpose of the above discussion was to give the reader some idea about how minimization of logic functions may be automated for use in CAD tools. We chose a scheme that is not too difficult to explain. From the practical point of view, this scheme has some drawbacks. The main difficulty is that the number of cubes that must be considered in the process can be extremely large.

If the goal of minimization is relaxed so that it is not imperative to find a minimum-cost implementation, then it is possible to derive heuristic techniques that produce good results in reasonable time. A technique of this type forms the basis of the widely used Espresso program, which is available on the Internet from the University of California at Berkeley. Espresso is a two-level optimization program. Both input to the program and its output are specified in the format of cubes. Instead of using the *-operation to find the prime implicants, one of the methods used in Espresso is an implicant-expansion technique. (See Problem 8.25 for an illustration of the expansion of implicants.) Many publications that explain the details used in the Espresso tool can be found in the technical literature.

The University of California at Berkeley has provided software programs that can be used for design of multilevel circuits, called MIS, SIS, and ABC. They allow a user to apply various multilevel optimization techniques to a logic circuit. The user can experiment with different optimization strategies by applying techniques such as factoring and decomposition to all or part of a circuit.

Numerous commercial CAD systems are on the market. Some companies whose products are widely used are Intel, Cadence Design Systems, Mentor Graphics, Synopsys, and Xilinx. Information on their products is available on the World Wide Web. Each company provides logic synthesis software that can be used to target various types of chips, such as FPGAs, gate arrays, standard cells, and custom chips. Because there are many possible ways to synthesize a given circuit, as we saw in the previous sections, each commercial product uses a proprietary logic optimization strategy based on heuristic methods.

To describe CAD tools, some new terminology has been invented. In particular, we should mention two terms that are widely used in industry: *technology-independent logic synthesis* and *technology mapping*. The first

term refers to techniques that are applied when optimizing a circuit without considering the resources available in the target chip. Most of the techniques presented in this chapter are of this type. The second term, technology mapping, refers to techniques that are used to ensure that the circuit produced by logic synthesis can be realized using the logic resources available in the target chip. A good example of technology mapping is the transformation from a circuit in the form of logic operations such as AND and OR into a circuit that consists of only NAND operations. This type of technology mapping is done when targeting a circuit to a gate array that contains only NAND gates. Another example is the translation from logic operations to lookup tables, which is done when targeting a design to an FPGA.

Chapter 10 discusses the CAD tools in detail. It presents a typical design flow that a designer may use to implement a digital system.

8.5 Concluding Remarks

This chapter has attempted to provide the reader with an understanding of various aspects of synthesis for logic functions. In the next chapter we discuss asynchronous sequential circuits. In the design of these circuits we will make use of many of the synthesis techniques covered in the previous chapters, as well as the cubical notation that we introduced in this chapter.

8.6 Examples of Solved Problems

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

Example 8.19

Problem: Consider the four-input logic function

$$f = x_1x_2x_4 + x_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_2\bar{x}_3$$

Show how this function can be implemented in an FPGA that has three-input LUTs.

Solution: A straightforward implementation requires 4 three-input LUTs. Three of these LUTs are used for the AND operations, and the final LUT implements the OR operation. But, if the function is synthesized as

$$f = x_2 \cdot (x_1 x_4 + x_3 \bar{x}_4) + \bar{x}_2 \cdot (\bar{x}_1 \bar{x}_3)$$

then only three LUTs are needed, as illustrated in Figure 8.33. The first LUT produces the subfunction $g = x_1 x_4 + x_3 \bar{x}_4$, the second LUT generates $h = \bar{x}_1 \bar{x}_3$, and the final LUT produces $f = x_2 \cdot g + \bar{x}_2 \cdot h$.

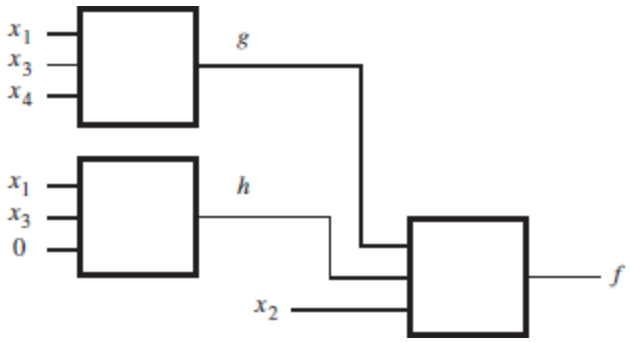


Figure 8.33 The function in Example 8.19.

Example 8.20

Problem: In Example 8.11 we created a BDD for the function $f = x_1 x_3 + x_1 x_4 + x_2 x_4 + x_2 x_3 + \bar{x}_1 \bar{x}_2 x_3 x_4$. Use functional decomposition to derive a multilevel implementation of f. Consider subfunctions of the inputs x_3 and x_4.

Solution: Figure 8.34 shows f in the form of a Karnaugh map. It has the subfunction $h = x_3 x_4$ in the first column, and the subfunction $g = x_3 + x_4$ in the second, third, and fourth columns. Hence

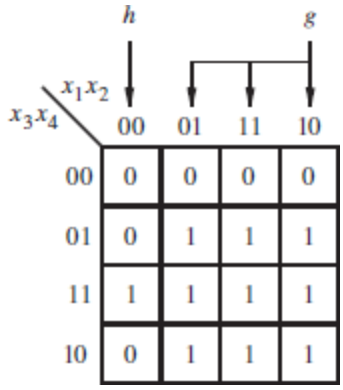


Figure 8.34 The function for Example 8.20.

$$\begin{aligned} f &= (\bar{x}_1\bar{x}_2)\cdot h + (\bar{x}_1x_2 + x_1x_2 + x_1\bar{x}_2)\cdot g \\ &= (\bar{x}_1\bar{x}_2)\cdot x_3x_4 + (x_1 + x_2)\cdot(x_3 + x_4) \end{aligned}$$

This is the same expression that was derived from the BDD in Example 8.11. It is possible to simplify this expression. Let $k = (x_1 + x_2)$, so that

$$\begin{aligned} f &= \bar{k}\cdot x_3x_4 + k\cdot(x_3 + x_4) \\ &= \bar{k}\cdot x_3x_4 + k\cdot x_3 + k\cdot x_4 \\ &= \bar{k}\cdot x_3x_4 + k\cdot x_3x_4 + k\cdot x_3 + k\cdot x_4 \\ &= (\bar{k} + k)\cdot x_3x_4 + k\cdot(x_3 + x_4) \\ &= x_3x_4 + k\cdot(x_3 + x_4) \end{aligned}$$

Here, we have used the absorption rule 13*a* from Section 2.5 to expand $k\cdot x_3$ into the two terms $k\cdot x_3x_4 + k\cdot x_3$, which then leads to a simplification. Resubstituting $k = (x_1 + x_2)$ gives

$$f = x_3x_4 + (x_1 + x_2)\cdot(x_3 + x_4)$$

We can also derive this decomposition of f by examining the rows of the Karnaugh map in Figure 8.34 and identifying subfunctions of variables x_1 and x_2. The second and fourth rows are the subfunction $k = x_1 + x_2$, and the third row is the subfunction equal to 1. Thus

$$\begin{aligned} f &= (\bar{x}_3x_4 + x_3\bar{x}_4)\cdot k + (x_3x_4)\cdot 1 \\ &= (\bar{x}_3x_4)\cdot k + (x_3\bar{x}_4)\cdot k + (x_3x_4)\cdot k + x_3x_4 \\ &= (\bar{x}_3x_4 + x_3\bar{x}_4 + x_3x_4)\cdot k + x_3x_4 \\ &= (x_3 + x_4)\cdot k + x_3x_4 \\ &= (x_3 + x_4)\cdot(x_1 + x_2) + x_3x_4 \end{aligned}$$

Example 8.21

Problem: Show how Shannon's expansion can be used to create a BDD for the function $f = x_1x_3 + x_2x_4$, with the input order x_1, x_2, x_3, x_4.

Solution: Shannon's expansion with respect to x_1 gives $f = \bar{x}_1 f_{\bar{x}_1} + x_1 f_{x_1}$. The cofactors are

$$f_{\bar{x}_1} = 0 \cdot x_3 + x_2x_4 = x_2x_4$$
$$f_{x_1} = 1 \cdot x_3 + x_2x_4 = x_3 + x_2x_4$$

For the cofactor $f_{x_1} = x_3 + x_2x_4$ it is useful to apply Shannon's expansion again, with respect to input x_2. This gives

$$f_{x_1\bar{x}_2} = x_3 + 0 \cdot x_4 = x_3$$
$$f_{x_1x_2} = x_3 + 1 \cdot x_4 = x_3 + x_4$$

The resulting BDD is shown in Figure 8.35. The BDD for $f_{\bar{x}_1} = x_2x_4$ is highlighted in red. The BDD for f_{x_1} is on the right-hand side of the figure, with the subfunction $f_{x_1\bar{x}_2} = x_3$ on the 0-edge from node x_2 and the subfunction for $f_{x_1x_2} = x_3 + x_4$ on the 1-edge from node x_2.

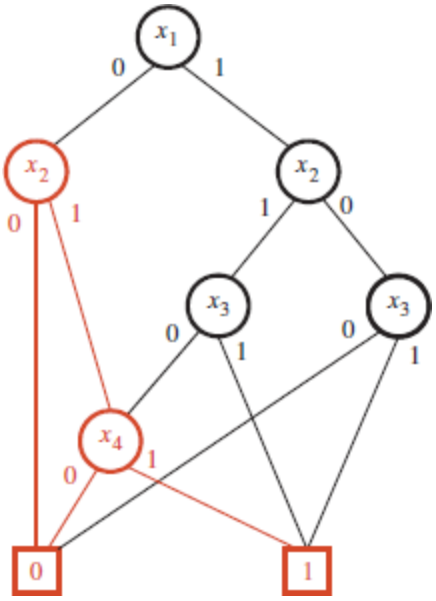
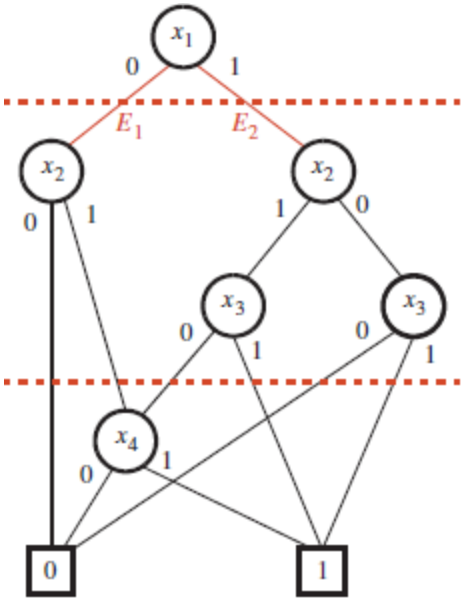


Figure 8.35 The BDD for Example 8.21.

Example 8.22

Problem: Create a BDD for the function used in Example 8.21 with the input order x_1, x_3, x_2, x_4.

Solution: We can derive the BDD by swapping the inputs x_2 and x_3 in the BDD of Figure 8.35. The first step is to isolate the part of the BDD that contains nodes x_2 and x_3, as shown by the dotted lines in Figure 8.36*a*. There are two edges, labeled E_1 and E_2, that enter this part of the BDD. There is no reordering needed for edge E_1, because it does not depend on node x_3. But for edge E_2, it is useful to create the truth table shown in Figure 8.36*b* to enumerate the paths in the BDD that start at E_2. The path representing $x_2x_3 = 00$ leads to the terminal node 0, the path for $x_2x_3 = 10$ leads to node x_4, and the remaining two paths lead to the terminal node 1.



(a) Isolating x_2 and x_3

x_2	x_3	E_2
0	0	0
0	1	1
1	0	x_4
1	1	1

(b) Truth table

Figure 8.36 Reordering the BDD in Figure 8.35.

We can now create the graph shown in Figure 8.37*a*, ensuring that all paths from edge E_2 reach the correct destination nodes shown in the truth table. Removing the redundant node x_2 on the right-hand side of this graph,

and also merging the two remaining identical copies of node x_2, results in the BDD in Figure 8.37*b*. This BDD is simpler than the one in Figure 8.35.

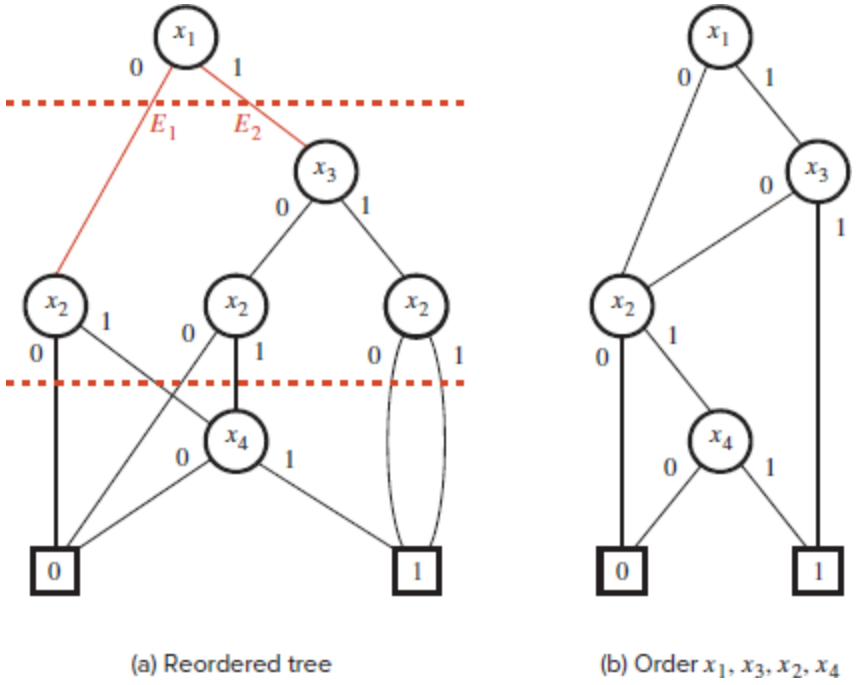


Figure 8.37 The BDD for Example 8.22.

Example 8.23

Problem: Use the tabular method of Section 8.4.1 to derive a minimum-cost SOP expression for the function

$$f(x_1, \ldots, x_4) = \bar{x}_1\bar{x}_3\bar{x}_4 + x_3x_4 + \bar{x}_1\bar{x}_2x_4 + x_1x_2\bar{x}_3x_4$$

assuming that there are also don't-cares defined as $D = \sum(9, 12, 14)$.

Solution: The tabular method requires that we start with the function defined in the form of minterms. As found in Figure 2.68, the function *f* can also be represented as

$$f(x_1, \ldots, x_4) = \sum m(0, 1, 3, 4, 7, 11, 13, 15) + D(9, 12, 14)$$

The corresponding eleven 0-cubes are placed in list 1 in Figure 8.38.

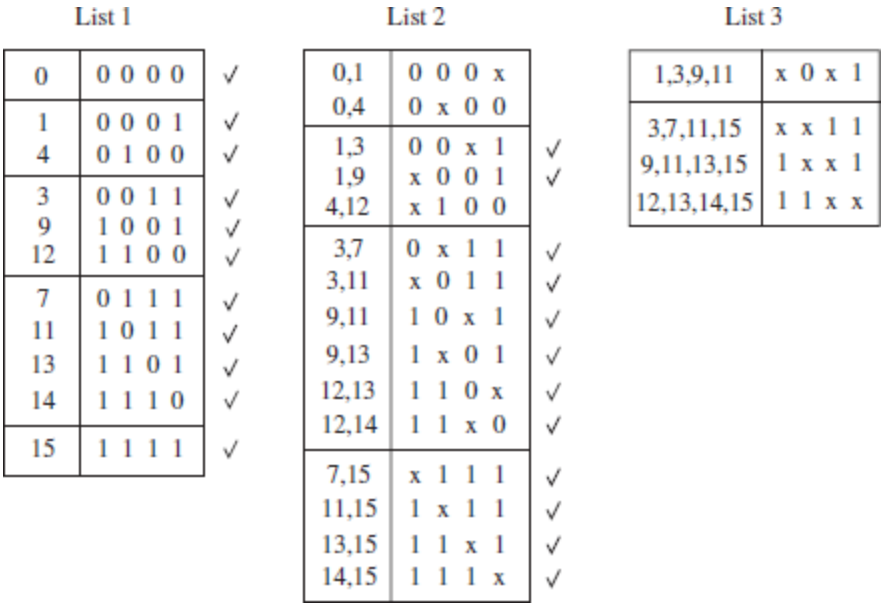


Figure 8.38 Generation of prime implicants for the function in Example 8.23.

Now, we perform a pairwise comparison of all 0-cubes to determine the 1-cubes shown in list 2, which are obtained by combining pairs of 0-cubes. Note that all 0-cubes are included in the 1-cubes, as indicated by the checkmarks in list 1. Next, perform a pairwise comparison of all 1-cubes to obtain the 2-cubes in list 3. Some of these 2-cubes can be generated in multiple ways, but it is not useful to list a 2-cube more than once (for example, x0x1 in list 3 can be obtained by combining from list 2 the cubes 1,3 and 9,11 or by using the cubes 1,9 and 3,11). Note that all but three 1-cubes are included in the 2-cubes. It is not possible to generate any 3-cubes, hence all terms that are not included in some other term (the unchecked terms in list 2 and all terms in list 3) are the prime implicants of *f*. The set of prime implicants is

$$P = \{000x, 0x00, x100, x0x1, xx11, 1xx1, 11xx\}$$
$$= \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$$

To find the minimum-cost cover for *f*, construct the table in Figure 8.39*a* which shows all prime implicants and the minterms that must be covered, namely those for which $f = 1$. A checkmark is placed to indicate that a minterm is covered by a particular prime implicant. Since minterm 7 is covered only by p_5, this prime implicant must be included in the final cover. Observe that row p_2 dominates row p_3, hence the latter can be removed.

Similarly, row p_6 dominates row p_7. Removing rows p_5, p_3, and p_7, as well as columns 3, 7, 11, and 15 (which are covered by p_5), leads to the reduced table in Figure 8.39*b*. In this table, p_2 and p_6 are essential. They cover minterms 0, 4, and 13. Thus, it remains only to cover minterm 1, which can be done by choosing either p_1 or p_4. Since p_4 has a lower cost, it should be chosen. Therefore, the final cover is

Prime implicant	Minterm 0	1	3	4	7	11	13	15
p_1 = 0 0 0 x	✓	✓						
p_2 = 0 x 0 0	✓			✓				
p_3 = x 1 0 0				✓				
p_4 = x 0 x 1		✓	✓			✓		
p_5 = x x 1 1			✓		✓	✓		✓
p_6 = 1 x x 1						✓	✓	✓
p_7 = 1 1 x x							✓	✓

(a) Initial prime implicant cover table

Prime implicant	Minterm 0	1	4	13
p_1 = 0 0 0 x	✓	✓		
p_2 = 0 x 0 0	✓		✓	
p_4 = x 0 x 1		✓		
p_6 = 1 x x 1				✓

(b) After the removal of rows p_3, p_5 and p_7 and columns 3, 7, 11 and 15

Figure 8.39 Selection of a cover for the function in Example 8.23.

$$C = \{p_2, p_4, p_5, p_6\}$$
$$= \{0x00, x0x1, xx11, 1xx1\}$$

and the function is implemented as

$$f = \bar{x}_1\bar{x}_3\bar{x}_4 + \bar{x}_2x_4 + x_3x_4 + x_1x_4$$

Example 8.24

Problem: Use the $*$-product operation to find all prime implicants of the function

$$f(x_1, \ldots, x_4) = \bar{x}_1\bar{x}_3\bar{x}_4 + x_3x_4 + \bar{x}_1\bar{x}_2x_4 + x_1x_2\bar{x}_3x_4$$

assuming that there are also don't-cares defined as $D = \sum(9, 12, 14)$.

Solution: The ON-set for this function is

$$\text{ON} = \{0x00, xx11, 00x1, 1101\}$$

The initial cover, consisting of the ON-set and the don't-cares, is

$$C^0 = \{0x00, xx11, 00x1, 1101, 1001, 1100, 1110\}$$

Using the *-operation, the subsequent covers obtained are

$$C^1 = \{0x00, xx11, 00x1, 000x, x100, 11x1, 10x1, 111x, x001, 1x01, 110x, 11x0\}$$
$$C^2 = \{0x00, xx11, 000x, x100, x0x1, 1xx1, 11xx\}$$
$$C^3 = C^2$$

Therefore, the set of all prime implicants is

$$P = \{\bar{x}_1\bar{x}_3\bar{x}_4, x_3x_4, \bar{x}_1\bar{x}_2\bar{x}_3, x_2\bar{x}_3\bar{x}_4, \bar{x}_2x_4, x_1x_4, x_1x_2\}$$

Example 8.25

Problem: Find the minimum-cost implementation for the function

$$f(x_1, \ldots, x_4) = \bar{x}_1\bar{x}_3\bar{x}_4 + x_3x_4 + \bar{x}_1\bar{x}_2x_4 + x_1x_2\bar{x}_3x_4$$

assuming that there are also don't-cares defined as $D = \sum(9, 12, 14)$.

Solution: This is the same function as used in Examples 8.23 and 8.24. In those examples, we found that the minimum-cost SOP implementation is

$$f = x_3x_4 + \bar{x}_1\bar{x}_3\bar{x}_4 + \bar{x}_2x_4 + x_1x_4$$

which requires four AND gates, one OR gate, and 13 inputs to the gates, for a total cost of 18.

The minimum-cost POS implementation is

$$f = (\bar{x}_3 + x_4)(\bar{x}_1 + x_4)(x_1 + \bar{x}_2 + x_3 + \bar{x}_4)$$

which requires three OR gates, one AND gate, and 11 inputs to the gates, for a total cost of 15.

We can also consider a multilevel realization for the function. Applying the factoring concept to the above SOP expression yields

$$f = (x_1 + \bar{x}_2 + x_3)x_4 + \bar{x}_1\bar{x}_3\bar{x}_4$$

This implementation requires two AND gates, two OR gates, and 10 inputs to the gates, for a total cost of 14. Compared to the SOP and POS implementations, this has the lowest cost in terms of gates and inputs, but it results in a slower circuit because there are three levels of gates through which the signals must propagate.

Example 8.26

Problem: In several commercial FPGAs the logic blocks are four-input LUTs. Two such LUTs, connected as shown in Figure 8.40, can be used to implement functions of seven variables by using the decomposition

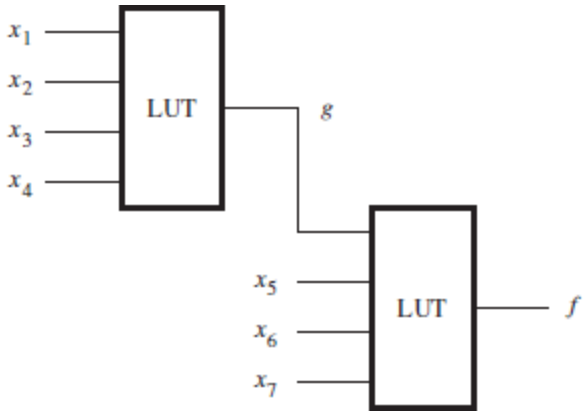


Figure 8.40 Circuit for Example 8.26.

$$f(x_1, \ldots, x_7) = f[g(x_1, \ldots, x_4), x_5, x_6, x_7]$$

It is easy to see that functions such as $f = x_1x_2x_3x_4x_5x_6x_7$ and $f = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$ can be implemented in this form. Show that there exist other seven-variable functions that cannot be implemented with 2 four-input LUTs.

Solution: The truth table for a seven-variable function can be arranged as depicted in Figure 8.41. There are $2^7 = 128$ minterms. Each valuation of the variables x_1, x_2, x_3, and x_4 selects one of the 16 columns in the truth table, while each valuation of x_5, x_6, and x_7 selects one of 8 rows. Since we have to use the circuit in Figure 8.40, the truth table for f can also be defined in terms of the subfunction g. In this case, it is g that selects one of the 16 columns in the truth table, instead of x_1, x_2, x_3, and x_4. Since g can have only two possible values, 0 and 1, we can have only two columns in the truth table. This is possible if there exist only two distinct patterns of 1s and 0s in the 16 columns in Figure 8.40. Therefore, only a relatively small subset of seven-variable functions can be realized with just two LUTs.

$x_5x_6x_7$ \ $x_1x_2x_3x_4$	0000	0001	...	1110	1111
000	m_0	m_8		m_{112}	m_{120}
001	m_1	m_9		m_{113}	m_{121}
010	m_2	m_{10}		m_{114}	m_{122}
011	m_3	m_{11}	...	m_{115}	m_{123}
100	m_4	m_{12}		m_{116}	m_{124}
101	m_5	m_{13}		m_{117}	m_{125}
110	m_6	m_{14}		m_{118}	m_{126}
111	m_7	m_{15}		m_{119}	m_{127}

Figure 8.41 A possible format for truth tables of seven-variable functions.

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

***8.1** Implement the logic circuit in Figure 8.6*b* using NAND gates only.

***8.2** Implement the logic circuit in Figure 8.6*b* using NOR gates only.

8.3 Implement the logic circuit in Figure 8.8*b* using NAND gates only.

8.4 Implement the logic circuit in Figure 8.8*b* using NOR gates only.

***8.5** Consider the function $f = x_3x_5 + \bar{x}_1x_2x_4 + x_1\bar{x}_2\bar{x}_4 + x_1x_3\bar{x}_4 + \bar{x}_1x_3x_4 + \bar{x}_1x_2x_5 + x_1\bar{x}_2x_5$ Derive a minimum-cost circuit that implements this function using NOT, AND, and OR gates.

8.6 Derive a minimum-cost circuit that implements the function $f(x_1, \ldots, x_4) = \sum m(4, 7, 8, 11) + D(12, 15)$.

8.7 Find the simplest realization of the function $f(x_1, \ldots, x_4) = \sum m(0, 3, 4, 7, 9, 10, 13, 14)$, assuming that the logic gates have a maximum fan-in of two.

***8.8** Find the minimum-cost circuit for the function $f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 13, 14, 15)$. Assume that the input variables are available in uncomplemented form only. (*Hint:* Use functional decomposition.)

8.9 Use functional decomposition to find the best implementation of the function $f(x_1, \ldots, x_5) = \sum m(1, 2, 7, 9, 10, 18, 19, 25, 31) + D(0, 15, 20, 26)$. How does your implementation compare with the lowest-cost SOP implementation? Give the costs.

8.10 In Figure 8.7 we decomposed the function *f* by first identifying subfunctions in the columns of the Karnaugh map. Repeat this decomposition by first identifying subfunctions in the rows of the Karnaugh map.

8.11 In Figure 8.8 we decomposed the function *f* by first identifying subfunctions in the rows of the Karnaugh map. Repeat this decomposition by first identifying subfunctions in the columns of the Karnaugh map.

***8.12** Create a BDD for the function in Figure 8.6 using the input order x_1, x_2, x_3, x_4.

8.13 Repeat Problem 8.12 using the input order x_4, x_3, x_2, x_1.

8.14 Create a BDD for the function in Figure 8.8 using the input order x_1, x_2, x_3, x_4, x_5.

8.15 Repeat Problem 8.14 using the input order x_5, x_4, x_3, x_2, x_1.

8.16 For the function $f = x_1x_3 + x_2x_4$ in Example 8.21 show how to use Shannon's expansion to derive the BDD in Figure 8.37*b*.

8.17 Show how to swap the variables x_2 and x_3 in the BDD in Figure 8.37*b* to derive the BDD in Figure 8.35.

***8.18** Use the tabular method discussed in Section 8.4.1 to find a minimum cost SOP realization for the function

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 7, 8, 9, 15)$$

8.19 Repeat Problem 8.18 for the function

$$f(x_1, \ldots, x_4) = \sum m(0, 4, 6, 8, 9, 15) + D(3, 7, 11, 13)$$

8.20 Repeat Problem 8.18 for the function

$$f(x_1, \ldots, x_4) = \sum m(0, 3, 4, 5, 7, 9, 11) + D(8, 12, 13, 14)$$

8.21 Show that the following distributive-like rules are valid

$$(A \cdot B)\#C = (A\#C) \cdot (B\#C)$$
$$(A + B)\#C = (A\#C) + (B\#C)$$

8.22 Use the cubical representation and the method discussed in Section 8.4.2 to find a minimum-cost SOP realization of the function $f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 7, 8, 9, 15)$.

8.23 Repeat Problem 8.22 for the function

$$f(x_1, \ldots, x_5) = \bar{x}_1\bar{x}_3\bar{x}_5 + x_1x_2\bar{x}_3 + x_2x_3\bar{x}_4x_5 + x_1\bar{x}_2\bar{x}_3x_4 + x_1x_2x_3x_4\bar{x}_5 + \bar{x}_1x_2x_4\bar{x}_5 + \bar{x}_1\bar{x}_3x_4x_5$$

8.24 Use the cubical representation and the method discussed in Section 8.4.2 to find a minimum-cost SOP realization of the function $f(x_1, \ldots, x_4)$ defined by the ON-set ON = {00x0, 100x, x010, 1111} and the don't-care set DC = {00x1, 011x}.

8.25 In Section 8.4.2 we showed how the $*$-product operation can be used to find the prime implicants of a given function *f*. Another possibility is to find the prime implicants by expanding the implicants in the initial cover of the function. An implicant is *expanded* by removing one literal to create a larger implicant (in terms of the number of vertices covered). A larger implicant is valid only if it does not include any vertices for which $f = 0$. The

largest valid implicants obtained in the process of expansion are the prime implicants. Figure P8.1 illustrates the expansion of the minterm $\bar{x}_1x_2x_3$ of the function used in Example 8.14. Note from Figure 2.56 that

$$\bar{f} = x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3$$

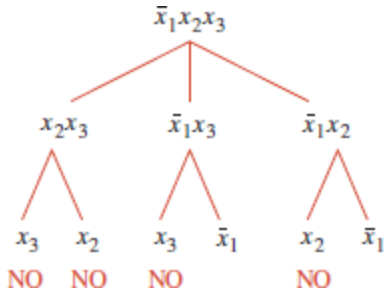


Figure P8.1 Expansion of implicant $\bar{x}_1x_2x_3$.

In Figure P8.1 the word NO is used to indicate that the expanded term is not valid, because it includes one or more vertices from $\bar{f}$. From the graph it is clear that the largest valid implicants that arise from this expansion are x_2x_3 and $\bar{x}_1$; they are prime implicants of *f*.

Expand the other four minterms given in the initial cover in Example 8.14 to find all prime implicants of *f*. What is the relative complexity of this procedure compared to the $*$-product technique?

8.26 Repeat Problem 8.25 for the function in Example 8.15. Expand the implicants given in the initial cover C^0.

8.27 Find the minimum-cost circuit consisting only of two-input NAND gates for the function $f(x_1, \ldots, x_4) = \sum m(0, 1, 2, 3, 4, 6, 8, 9, 12)$. Assume that the input variables are available in both uncomplemented and complemented forms. (*Hint:* Consider the complement of the function.)

8.28 Repeat Problem 8.27 for the function $f(x_1, \ldots, x_4) = \sum m(2, 3, 6, 8, 9, 12)$.

8.29 Find the minimum-cost circuit consisting only of two-input NOR gates for the function $f(x_1, \ldots, x_4) = \sum m(6, 7, 8, 10, 12, 14, 15)$. Assume that the input variables are available in both uncomplemented and complemented forms. (*Hint:* Consider the complement of the function.)

8.30 Repeat Problem 8.29 for the function $f(x_1, \ldots, x_4) = \sum m(2, 3, 4, 5, 9, 10, 11, 12, 13, 15)$.

chapter

9

ASYNCHRONOUS SEQUENTIAL CIRCUITS

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Sequential circuits that are not synchronized by a clock
- Analysis of asynchronous sequential circuits
- Synthesis of asynchronous sequential circuits
- The concept of stable and unstable states
- Hazards that cause incorrect behavior of a circuit
- Timing issues in digital circuits

In Chapter 6 we covered the design of synchronous sequential circuits in which the state variables are represented by flip-flops that are controlled by a clock. The clock is a periodic signal that consists of pulses. Changes in state can occur on the positive or negative edge of each clock pulse. Since

they are controlled by pulses, synchronous sequential circuits are said to operate in *pulse mode*. In this chapter we present sequential circuits that do not operate in pulse mode and do not use flip-flops to represent state variables. These circuits are called *asynchronous sequential circuits*.

In an asynchronous sequential circuit, changes in state are not triggered by clock pulses. Instead, changes in state are dependent on whether each of the inputs to the circuit has the logic level 0 or 1 at any given time. To achieve reliable operation, the inputs to the circuit must change in a specific manner. In this introductory discussion we will concentrate on the simplest case in which a constraint is imposed that the inputs must change one at a time. Moreover, there must be sufficient time between the changes in input signals to allow the circuit to reach a *stable state*, which is achieved when all internal signals stop changing. A circuit that adheres to these constraints is said to operate in the *fundamental mode*.

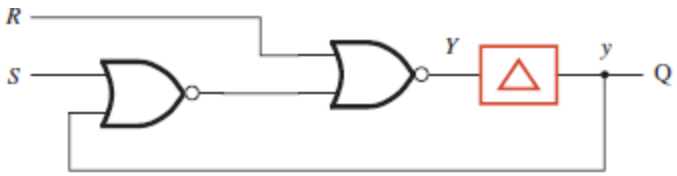
Asynchronous circuits are much more difficult to design than synchronous circuits. Specialized techniques, which are beyond the scope of this book, have been developed for dealing with large asynchronous circuits. Our main reason for the discussion in this chapter is the fact that the asynchronous circuits, even in their simplest form, provide an excellent vehicle for gaining a deeper understanding of the operation of digital circuits in general. In particular, they illustrate the timing issues caused by propagation delays in logic circuits.

The design approaches presented in this chapter are classical techniques that are suitable only for very small circuits. They are easy to understand and they demonstrate the problems that arise from timing constraints. In synchronous circuits these problems are avoided by using a clock as a synchronizing mechanism.

9.1 Asynchronous Behavior

To introduce asynchronous sequential circuits, we will reconsider the basic latch circuit in Figure 5.3. This Set-Reset (SR) latch is redrawn in Figure 9.1*a*. The feedback loop gives rise to the sequential nature of the circuit. It is an asynchronous circuit because changes in the value of the output, Q, occur without having to wait for a synchronizing clock pulse. In response to

a change in either the *S* (Set) or *R* (Reset) input, the value of Q will change after a short propagation time through the NOR gates. In Figure 9.1*a* the combined propagation delay through the two NOR gates is represented by the box labeled Δ. Then, the NOR gate symbols represent ideal gates with zero delay. Using the notation in Chapter 6, Q corresponds to the *present state* of the circuit, represented by the *present-state variable*, *y*. The value of *y* is fed back through the circuit to generate the value of the *next-state variable*, *Y*, which represents the *next state* of the circuit. After the Δ time delay, *y* takes the value of *Y*. Observe that we have drawn the circuit in a style that conforms to the general model for sequential circuits presented in Figure 6.87.



(a) Circuit with modeled gate delay

Present state y	Next state			
	SR = 00	01	10	11
	Y	Y	Y	Y
0	(0)	(0)	1	(0)
1	(1)	0	(1)	0

(b) State-assigned table

Figure 9.1 Analysis of the SR latch.

By analyzing the SR latch, we can derive a state-assigned table, as illustrated in Figure 9.1*b*. When the present state is $y = 0$ and the inputs are $S = R = 0$, the circuit produces $Y = 0$. Since $y = Y$, the state of the circuit will not change. We say that the circuit is *stable* under these input conditions. Now assume that *R* changes to 1 while *S* remains at 0. The circuit still generates $Y = 0$ and remains stable. Assume next that *S* changes to 1 and *R* remains at 1. The value of *Y* is unchanged, and the circuit is stable. Then let *R* change to 0 while *S* remains at 1. This input valuation, $SR = 10$, causes the circuit to generate $Y = 1$. Since $y \neq Y$, the circuit is not

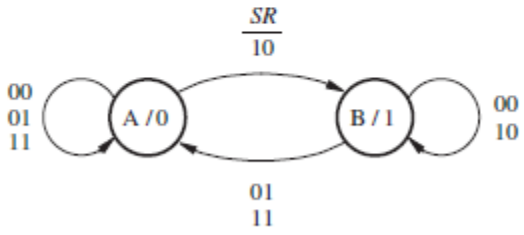
stable. After the Δ time delay, the circuit changes to the new present state $y = 1$. Once this new state is reached, the value of Y remains equal to 1 as long as $SR = 10$. Hence the circuit is again stable. The analysis for the present state $y = 1$ can be completed using similar reasoning.

The concept of stable states is very important in the context of asynchronous sequential circuits. For a given valuation of inputs, if a circuit reaches a particular state and remains in this state, then the state is said to be stable. To clearly indicate the conditions under which the circuit is stable, it is customary to encircle the stable states in the table, as illustrated in Figure 9.1*b*.

From the state-assigned table, we can derive the state table in Figure 9.2*a*. The state names A and B represent the present states $y = 0$ and $y = 1$, respectively. Since the output Q depends only on the present state, the circuit is a Moore-type FSM. The state diagram that represents the behavior of this FSM is shown in Figure 9.2*b*.

Present state	Next state				Output Q
	$SR = 00$	01	10	11	
A	(A)	(A)	B	(A)	0
B	(B)	A	(B)	A	1

(a) State table



(b) State diagram

Figure 9.2 FSM model for the SR latch.

The preceding analysis shows that the behavior of an asynchronous sequential circuit can be represented as an FSM in a similar way as the synchronous sequential circuits in Chapter 6. Consider now performing the opposite task. That is, given the state table in Figure 9.2*a*, we can synthesize an asynchronous circuit as follows: After performing the state assignment, we have the state-assigned table in Figure 9.1*b*. This table

represents a truth table for Y, with the inputs y, S, and R. Deriving a minimal product-of-sums expression yields

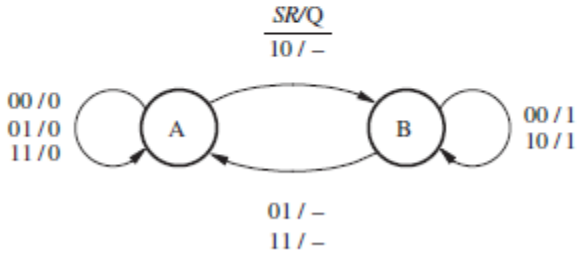
$$Y = \overline{R} \cdot (S + y)$$

If we were deriving a synchronous sequential circuit using the methods in Chapter 6, then Y would be connected to the D input of a flip-flop and a clock signal would be used to control the time when the changes in state take place. But since we are synthesizing an asynchronous circuit, we do not insert a flip-flop in the feedback path. Instead, we create a circuit that realizes the preceding expression using the necessary logic gates, and we feed back the output signal as the present-state input y. Implementation using NOR gates results in the circuit in Figure 9.1*a*. This simple example suggests that asynchronous circuits and synchronous circuits can be synthesized using similar techniques. However, we will see shortly that for more complex asynchronous circuits, the design task is considerably more difficult.

To further explore the nature of asynchronous circuits, it is interesting to consider how the behavior of the SR latch can be represented in the form of a Mealy model. As depicted in Figure 9.3, the outputs produced when the circuit is in a stable state are the same as in the Moore model, namely 0 in state A and 1 in state B. Consider now what happens when the state of the circuit changes. Suppose that the present state is A and that the input valuation SR changes from 00 to 10. As the state table specifies, the next state of the FSM is B. When the circuit reaches state B, the output Q will be 1. But in the Mealy model, the output is supposed to be affected immediately by a change in the input signals. Thus while still in state A, the change in SR to 10 should result in Q = 1. We could have written a 1 in the corresponding entry in the top row of the state table, but we have chosen to leave this entry unspecified instead. The reason is that since Q will change to 1 as soon as the circuit reaches state B, there is little to be gained in trying to make Q go to 1 a little sooner. Leaving the entry unspecified allows us to assign either 0 or 1 to it, which may make the circuit that implements the state table somewhat simpler. A similar reasoning leads to the conclusion that the two output entries where a change from B to A takes place can also be left unspecified.

Present state	Next state				Output, Q			
	SR = 00	01	10	11	00	01	10	11
A	(A)	(A)	B	(A)	0	0	–	0
B	(B)	A	(B)	A	1	–	1	–

(a) State table



(b) State diagram

Figure 9.3 Mealy representation of the SR latch.

Using the state assignment $y = 0$ for A and $y = 1$ for B, the state-assigned table represents a truth table for both Y and Q. The minimal expression for Y is the same as for the Moore model. To derive an expression for Q, we need to set the unspecified entries to 0 or 1. Assigning a 0 to the unspecified entry in the first row and 1 to the two unspecified entries in the second row produces $Q = y$ and results in the circuit in Figure 9.1*a*.

Terminology

In the preceding discussion we used the same terminology as in the previous chapter on synchronous sequential circuits. However, when dealing with asynchronous sequential circuits, it is customary to use two different terms. Instead of a "state table," it is more common to speak of a *flow table*, which indicates how the changes in state flow as a result of the changes in the input signals. Instead of a "state-assigned table," it is usual to refer to a *transition table* or an *excitation table*. We will use the terms *flow table* and *excitation table* in this chapter. A flow table will define the state changes and outputs that must be generated. An excitation table will depict the transitions in terms of the state variables. The term excitation table derives from the fact that a change from a stable state is performed by "exciting" the next-state variables to start changing towards a new state.

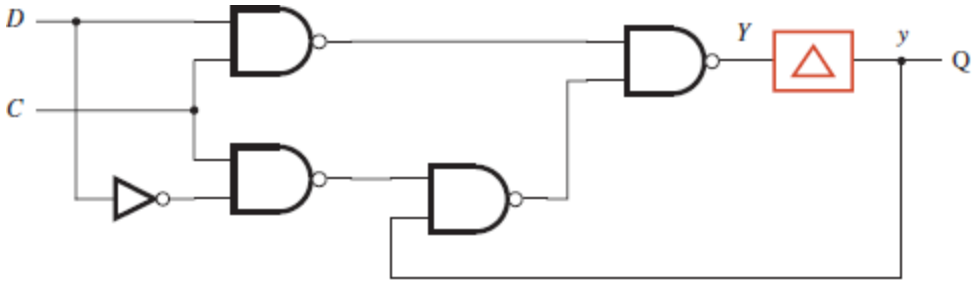
9.2 Analysis of Asynchronous Circuits

To gain familiarity with asynchronous circuits, it is useful to analyze a few examples. We will keep in mind the general model in Figure 6.87, assuming that the delays in the feedback paths are a representation of the propagation delays in the circuit. Then each gate symbol will represent an ideal gate with zero delay.

Example 9.1

GATED D LATCH In Chapters 5 and 6, we used the gated D latch as a key component in circuits that are controlled by a synchronizing clock. It is instructive to analyze this latch as an asynchronous circuit, where the clock is just one of the inputs. It is reasonable to assume that the signals on the *D* and clock inputs do not change at the same time, thus meeting the basic requirement of asynchronous circuits.

Figure 9.4*a* shows the gated D latch drawn in the style of the model of Figure 6.87. This circuit was introduced in Figure 5.7 and discussed in Section 5.3. The next-state expression for this circuit is



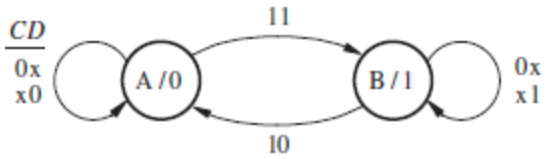
(a) Circuit

Present state	Next state				
	CD = 00	01	10	11	
y	Y	Y	Y	Y	Q
0	(0)	(0)	(0)	1	0
1	(1)	(1)	0	(1)	1

(b) Excitation table

Present state	Next state				
	CD = 00	01	10	11	Q
A	(A)	(A)	(A)	B	0
B	(B)	(B)	A	(B)	1

(c) Flow table



(d) State diagram

Figure 9.4 The gated D latch.

$$Y = (C \uparrow D) \uparrow ((C \uparrow \overline{D}) \uparrow y)$$
$$= CD + \overline{C}y + Dy$$

The term $D\,y$ in this expression is redundant and could be deleted without changing the logic function of Y. Hence the minimal expression is

$$Y = CD + \overline{C}y$$

The reason that the circuit implements the redundant term Dy is that this term solves a race condition known as a *hazard*; we will discuss hazards in detail in Section 9.6.

Evaluating the expression for Y for all valuations of C, D, and y leads to the excitation table in Figure 9.4*b*. Note that the circuit changes its state only when $C = 1$ and D is different from the present state, y. In all other cases the circuit is stable. Using the symbols A and B to represent the states $y = 0$ and $y = 1$, we obtain the flow table and the state diagram shown in parts (*c*) and (*d*).

Example 9.2

M-S D FLIP-FLOP In Example 9.1 we analyzed the gated D latch as an asynchronous circuit. Actually, all practical circuits are asynchronous. However, if the circuit's behavior is tightly controlled by a clock signal, then simpler operating assumptions can be used, as we did in Chapter 6. Recall that in a synchronous sequential circuit all signals change values in synchronization with the clock signal. Now we will analyze another synchronous circuit as if it were an asynchronous circuit.

Two gated D latches are used to implement the M-S D flip-flop, as illustrated in Figure 5.9. This circuit is reproduced in Figure 9.5. We can analyze the circuit by treating it as a series connection of two gated D latches. Using the results from Example 9.1, the simplified next-state expressions can be written as

$$Y_m = CD + \overline{C}y_m$$
$$Y_s = \overline{C}y_m + Cy_s$$

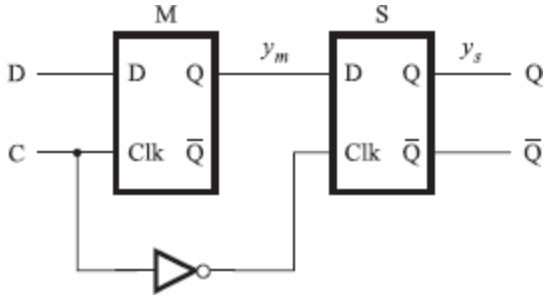


Figure 9.5 Circuit for the M-S D flip-flop.

where the subscripts *m* and *s* refer to the M- and S- stages of the flip-flop. These expressions lead to the excitation table in Figure 9.6*a*. Labeling the four states as *S*1 through *S*4, we derive the flow table in Figure 9.6*b*. A state-diagram form of this information is given in Figure 9.7.

Present state $y_m y_s$	Next state $Y_m Y_s$				Output Q
	$CD = 00$	01	10	11	
00	(00)	(00)	(00)	10	0
01	00	00	(01)	11	1
10	11	11	00	(10)	0
11	(11)	(11)	01	(11)	1

(a) Excitation table

Present state	Next state				Output Q
	$CD = 00$	01	10	11	
S1	(S1)	(S1)	(S1)	S3	0
S2	S1	S1	(S2)	S4	1
S3	S4	S4	S1	(S3)	0
S4	(S4)	(S4)	S2	(S4)	1

(b) Flow table

Present state	Next state				Output Q
	$CD = 00$	01	10	11	
S1	(S1)	(S1)	(S1)	S3	0
S2	S1	–	(S2)	S4	1
S3	–	S4	S1	(S3)	0
S4	(S4)	(S4)	S2	(S4)	1

(c) Flow table with unspecified entries

Figure 9.6 Excitation and flow tables for Example 9.2.

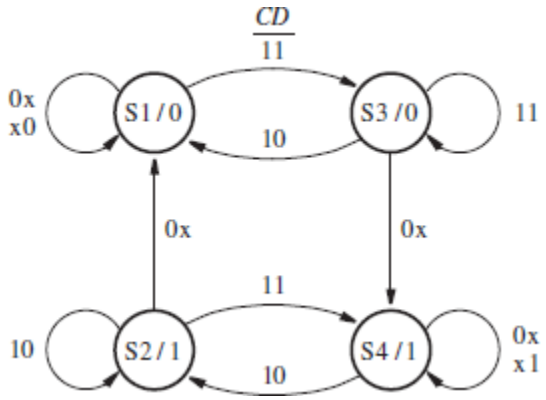


Figure 9.7 State diagram for the M-S D flip-flop.

Let us consider the behavior of this FSM in more detail. The state *S*1, where $y_m y_s = 00$, is stable for all input valuations except *CD* = 11. When *C* = 1, the value of *D* is stored in the M-stage; hence *CD* = 11 causes the flip-flop to change to *S*3, where $y_m = 1$ and $y_s = 0$. If the *D* input now changes back to 0, while the clock remains at 1, the flip-flop moves back to the state *S*1. The transitions between *S*1 and *S*3 indicate that if *C* = 1, the output of the M-stage, $Q_m = y_m$, tracks the changes in the *D* input signal without affecting the S-stage. From *S*3 the circuit changes to *S*4 when the clock goes to 0. In *S*4 both M- and S- stages are set to 1 because the information from the M-stage is transferred to the S-stage on the negative edge of the clock. Now the flip-flop remains in *S*4 until the clock goes to 1 and the *D* input changes to 0, which causes a change to *S*2. In *S*2 the M-stage is cleared to 0, but the S-stage remains at 1. Again the flip-flop may change between *S*2 and *S*4 because the M-stage will track the changes in the *D* input signal while *C* = 1. From *S*2 the circuit changes to *S*1 when the clock goes low.

In Figures 9.6 and 9.7, we indicated that the flip-flop has only one output Q, which one sees when the circuit is viewed as a negative-edge-triggered flip-flop. From the observer's point of view, the flip-flop has only two states, 0 and 1. But internally, the flip-flop consists of the M- and S- parts, which gives rise to the four states described above.

We should also examine the basic assumption that the inputs must change one at a time. If the circuit is stable in state *S*2, for which *CD* = 10, it is impossible to go from this state to *S*1 under the influence of the input valuation *CD* = 01 because this simultaneous change in both inputs cannot occur. Thus in the second row of the flow table, instead of showing *S*2 changing to *S*1 under *CD* = 01, this entry can be labeled as unspecified. The

change from $S2$ to $S1$ can be caused only by CD changing from 10 to 00. Similarly, if the circuit is in state $S3$, where $CD = 11$, it cannot change to $S4$ by having $CD = 00$. This entry can also be left unspecified in the table. The resulting flow table is shown in Figure 9.6*c*.

If we reverse the analysis procedure and, using the state assignment in Figure 9.6*a*, synthesize logic expressions for Y_m and Y_s, we get

$$Y_m = CD + \overline{C}y_m + y_m D$$
$$Y_s = \overline{C}y_m + Cy_s + y_m y_s$$

The terms $y_m D$ and $y_m y_s$ in these expressions are redundant. As mentioned earlier, they are included in the circuit to avoid race conditions, which are discussed in Section 9.6.

Example 9.3

Consider the circuit in Figure 9.8. It is represented by the following expressions

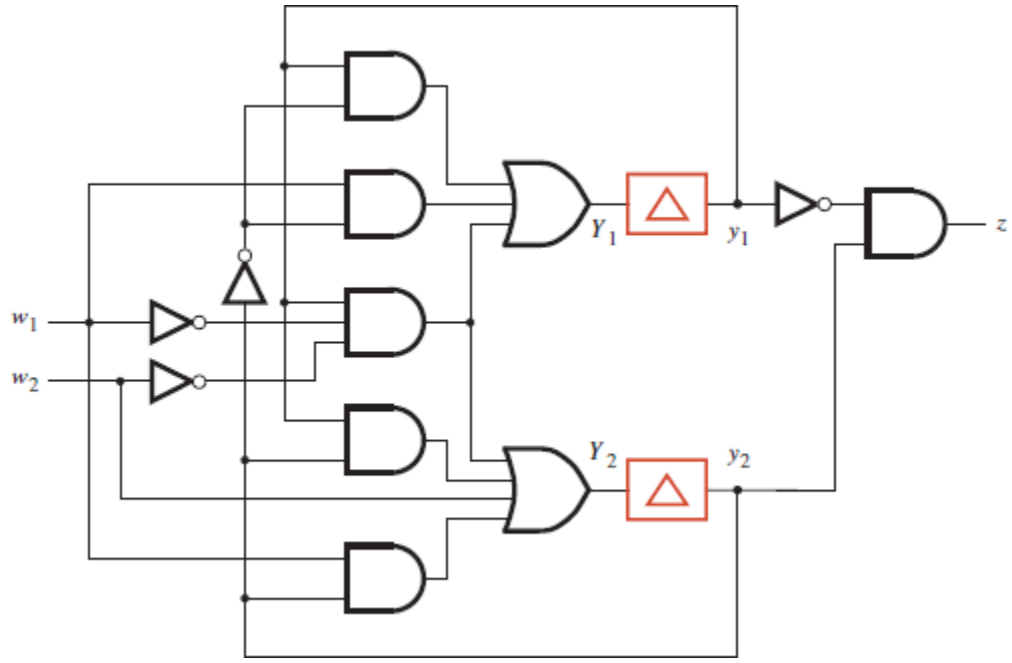


Figure 9.8 Circuit for Example 9.3.

$$Y_1 = y_1\bar{y}_2 + w_1\bar{y}_2 + \bar{w}_1\bar{w}_2y_1$$
$$Y_2 = y_1y_2 + w_1y_2 + w_2 + \bar{w}_1\bar{w}_2y_1$$
$$z = \bar{y}_1y_2$$

The corresponding excitation and flow tables are given in Figure 9.9.

Present state y_2y_1	Next state w_2w_1 = 00	01	10	11	Output z
	Y_2Y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	
00	(00)	01	10	11	0
01	11	(01)	11	11	0
10	00	(10)	(10)	(10)	1
11	(11)	10	10	10	0

(a) Excitation table

Present state	Next state w_2w_1 = 00	01	10	11	Output z
A	(A)	B	C	D	0
B	D	(B)	D	D	0
C	A	(C)	(C)	(C)	1
D	(D)	C	C	C	0

(b) Flow table

Figure 9.9 Excitation and flow tables for the circuit in Figure 9.8.

Some transitions in the flow table will not occur in practice because of the assumption that both w_1 and w_2 cannot change simultaneously. In state A the circuit is stable under the valuation $w_2\ w_1$ = 00. Its inputs cannot change to 11 without passing through the valuations 01 or 10, in which case the new state would be B or C, respectively. Thus the transition from A under $w_2\ w_1$ = 11 can be left unspecified. Similarly, if the circuit is stable in state B, in which case $w_2\ w_1$ = 01, it is impossible to force a change to state D by changing the inputs to $w_2\ w_1$ = 10. This entry should also be unspecified. If the circuit is stable in state C under $w_2\ w_1$ = 11, it is not possible to go to A by changing the inputs directly to $w_2\ w_1$ = 00. However,

the transition to *A* is possible by changing the inputs one at a time because the circuit remains stable in *C* for both $w_2\ w_1 = 01$ and $w_2\ w_1 = 10$

A different situation arises if the circuit is stable in state *D* under $w_2\ w_1 = 00$. It may seem that the entry under $w_2\ w_1 = 11$ should be unspecified because this input change cannot be made from the stable state *D*. But suppose that the circuit is stable in state *B* under $w_2\ w_1 = 01$. Now let the inputs change to $w_2\ w_1 = 11$. This causes a change to state *D*. The circuit indeed changes to *D*, but it is not stable in this state for this input condition. As soon as it arrives into state *D*, the circuit proceeds to change to state *C* as required by $w_2\ w_1 = 11$. It is then stable in state *C* as long as both inputs remain at 1. The conclusion is that the entry that specifies the change from *D* to *C* under $w_2\ w_1 = 11$ is meaningful and should not be omitted. The transition from the stable state *B* to the stable state *C*, which passes through state *D*, illustrates that it is not imperative that all transitions be directly from one stable state to another. A state through which a circuit passes en route from one stable state to another is called an *unstable state*. Transitions that involve passing through an unstable state are not harmful as long as the unstable state does not generate an undesirable output signal. For example, if a transition is between two stable states for which the output signal should be 0, it would be unacceptable to pass through an unstable state that causes the output to be 1. Even though the circuit changes through the unstable state very quickly, the short glitch in the output signal is likely to be troublesome. This is not a problem in our example. When the circuit is stable in *B*, the output is $z = 0$. When the inputs change to $w_2\ w_1 = 11$, the transition to state *D* maintains the output at 0. It is only when the circuit finally changes into state *C* that *z* will change to 1. Therefore, the change from $z = 0$ to $z = 1$ occurs only once during the course of these transitions.

A modified flow table, showing the unspecified transitions, is presented in Figure 9.10. The table indicates the behavior of the circuit in Figure 9.8 in terms of state transitions. If we don't know what the circuit is supposed to do, it may be difficult to discover the practical application for a given circuit. Fortunately, in practice the purpose of the circuit is known, and the analysis is done by the designer to ascertain that the circuit performs as desired. In our example it is apparent that the circuit generates the output $z = 1$ in state *C*, which it reaches as a result of some input patterns that are

detected using the other three states. The state diagram derived from Figure 9.10 is shown in Figure 9.11.

Present state	Next state				Output
	w_2w_1 = 00	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	D	0
C	A	(C)	(C)	(C)	1
D	(D)	C	C	C	0

Figure 9.10 Modified flow table for Example 9.3.

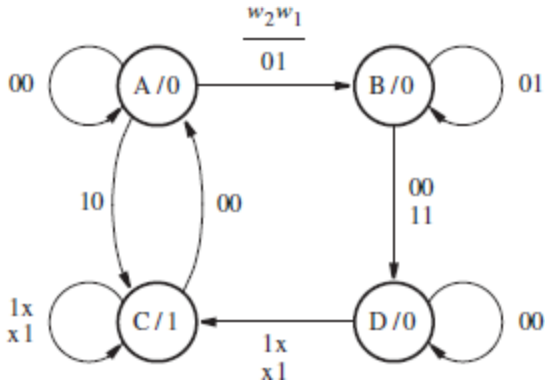


Figure 9.11 State diagram for Example 9.3.

This diagram actually implements a control mechanism for a simple vending machine that accepts two types of coins, say, dimes and nickels, and dispenses merchandise such as candy. If w_1 represents a nickel and w_2 represents a dime, then a total of 10 cents must be deposited to get the FSM into state *C* where the candy is released. The coin mechanism accepts only one coin at a time, which means that $w_2\ w_1 = 11$ can never occur. Therefore, the transition discussed above, from *B* to *C*, through the unstable state *D* would not occur. Observe that both states *B* and *D* indicate that 5 cents has been deposited. State *B* indicates that a nickel is presently being sensed by the coin receptor, while *D* indicates that 5 cents has been deposited and the coin receptor is presently empty. In state *D* it is possible to deposit either a nickel or a dime, both leading to state *C*. No distinction is made between the two types of coins in state *D*; hence the machine would not give change if 15 cents is deposited. From state *A* a dime leads directly to state *C*.

Knowing that the condition $w_2\, w_1 = 11$ will not occur allows the flow table to be specified as shown in Figure 9.12. If we were to synthesize the sum-of-products logic expressions for Y_1 and Y_2, using the state assignment in Figure 9.9*a*, we would end up with the circuit in Figure 9.8.

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	A	(C)	(C)	–	1
D	(D)	C	C	–	0

$w_2 \equiv$ dime $\quad w_1 \equiv$ nickel

Figure 9.12 Flow table for a simple vending machine.

Steps in the Analysis Process

We have demonstrated the analysis process using illustrative examples. The required steps can be stated as follows:

- A given circuit is interpreted in the form of the general model in Figure 6.87. That is, each feedback path is cut, and a delay element is inserted at the point where the cut is made. The input signal to the delay element represents a corresponding next-state variable, Y_i, while the output signal is the present-state variable, y_i. A cut can be made anywhere in a particular loop formed by the feedback connection, as long as there is only one cut per (state variable) loop. Thus the number of cuts that should be made is the smallest number that results in there being no feedback anywhere in the circuit except from the output of a delay element. This minimal number of cuts is sometimes referred to as the *cut set*. Note that the analysis based on a cut made at one point in a given loop may not produce the same flow table as an analysis on a cut made at some other point in this loop. But both flow tables would reflect the same functional behavior in terms of the applied inputs and generated outputs.
- Next-state and output expressions are derived from the circuit.

- The excitation table corresponding to the next-state and output expressions is derived.
- A flow table is obtained, associating some (arbitrary) names with the particular encoded states.
- A corresponding state diagram is derived from the flow table if desired.

9.3 Synthesis of Asynchronous Circuits

Synthesis of asynchronous sequential circuits follows the same basic steps used to synthesize the synchronous circuits, which were discussed in Chapter 6. There are some differences due to the asynchronous nature, which make the asynchronous circuits more difficult to design. We will explain the differences by investigating a few design examples. The basic steps are

- Devise a state diagram for an FSM that realizes the required functional behavior.
- Derive the flow table and reduce the number of states if possible.
- Perform the state assignment and derive the excitation table.
- Obtain the next-state and output expressions.
- Construct a circuit that implements these expressions.

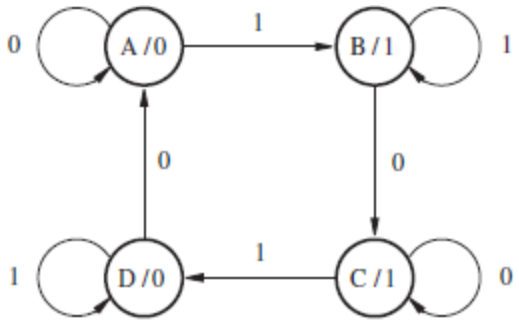
When devising a state diagram, or perhaps the flow table directly, it is essential to ensure that when the circuit is in a stable state, the correct output signals are generated. Should it be necessary to pass through an unstable state, this state must not produce an undesirable output signal. Minimization of states is not straightforward. A minimization procedure is described in Section 9.4.

State assignment is not done with the sole purpose of reducing the cost of the final circuit. In asynchronous circuits some state assignments may cause the circuit to be unreliable. We will explain this problem using the examples that follow.

Example 9.4

SERIAL PARITY GENERATOR Suppose that we want to design a circuit that has an input w and an output z, such that when pulses are applied to w, the output z is equal to 0 if the number of previously applied pulses is even and z is equal to 1 if the number of pulses is odd. Hence the circuit acts as a serial parity generator.

Let A be the state that indicates that an even number of pulses has been received. Using the Moore model, the output z will be equal to 0 when the circuit is in state A. As long as $w = 0$, the circuit should remain in A, which is specified by a transition arc that both originates and terminates in state A. Thus A is stable when $w = 0$. When the next pulse arrives, the input $w = 1$ should cause the FSM to move to a new state, say, B, which produces the output $z = 1$. When the FSM reaches B, it must remain stable in this state as long as $w = 1$. This is specified by a transition arc that originates and terminates in B. The next input change occurs when w goes to 0. In response the FSM must change to a state where $z = 1$ and which corresponds to the fact that a complete pulse has been observed, namely, that w has changed from 1 to 0. Let this state be C; it must be stable under the input condition $w = 0$. The arrival of the next pulse makes $w = 1$, and the FSM must change to a state, D, that indicates that an even number of pulses has been observed and that the last pulse is still present. The state D is stable under $w = 1$, and it causes the output to be $z = 0$. Finally, when w returns to 0 at the end of the pulse, the FSM returns to state A, which indicates an even number of pulses and w equal to 0 at the present time. The resulting state diagram is shown in Figure 9.13*a*.



(a) State diagram

Present state	Next state $w = 0$	Next state $w = 1$	Output z
A	(A)	B	0
B	C	(B)	1
C	(C)	D	1
D	A	(D)	0

(b) Flow table

Figure 9.13 Parity-generating asynchronous FSM.

A key point to understand is why it is necessary to have four states rather than just two, considering that we are merely trying to distinguish between the even and odd number of input pulses. States *B* and *C* cannot be combined into a single state even though they both indicate that an odd number of pulses has been observed. Suppose we had simply tried to use state *B* alone for this purpose. Then it would have been necessary to add an arc with a label 0 that originates and terminates in state *B*, which is fine. The problem is that without state *C*, there would have to be a transition from state *B* directly to *D* if the input is $w = 1$ to respond to the next change in the input when a new pulse arrives. It would be impossible to have *B* both stable under $w = 1$ and have a change to *D* effected for the same input condition. Similarly, we can show that the states *A* and *D* cannot be combined into a single state.

Figure 9.13*b* gives the flow table that corresponds directly to the state diagram. In many cases the designer can derive a flow table directly. We are using the state diagram mostly because it provides a simpler visual picture of the effect of the transitions in an FSM.

The next step is to assign values to the states in terms of the state variables. Since there are four states in our FSM, there have to be at least two state variables. Let these variables be y_1 and y_2. As a first attempt at the state assignment, let the states *A*, *B*, *C*, and *D* be encoded as $y_2\ y_1$ = 00, 01, 10, and 11, respectively. This assignment leads to the excitation table in Figure 9.14*a*. Unfortunately, it has a major flaw. The circuit that implements this table is stable in state *D* = 11 under the input condition *w* = 1. But consider what happens next if the input changes to *w* = 0. According to the excitation table, the circuit should change to state *A* = 00 and remain stable in this state. The problem is that in going from $y_2\ y_1$ = 11 to $y_2\ y_1$ = 00 both state variables must change their values. This is unlikely to occur at exactly the same time. In an asynchronous circuit the values of the next-state variables are determined by networks of logic gates with varying propagation delays. Thus we should expect that one state variable will change slightly before the other, which could put the circuit into a state where it may react to the input in an undesirable way. Suppose that y_1 changes first. Then the circuit goes from $y_2\ y_1$ = 11 to $y_2\ y_1$ = 10. As soon as it reaches this state, *C*, it will attempt to remain there if *w* = 0, which is a wrong outcome. On the other hand, suppose that y_2 changes first. Then there will be a change from $y_2\ y_1$ = 11 to $y_2\ y_1$ = 01, which corresponds to state *B*. Since *w* = 0, the circuit will now try to change to $y_2\ y_1$ = 10. This again requires that both y_1 and y_2 change; assuming that y_1 changes first in the transition from y_2y_1 = 01, the circuit will find itself in the state $y_2\ y_1$ = 00, which is the correct destination state, *A*. This discussion indicates that the required transition from *D* to *A* will be performed correctly if y_2 changes before y_1, but it will not work if y_1 changes before y_2. The result depends on the outcome of the “race” to change between the signals y_1 and y_2.

Present state y_2y_1	Next state Y_2Y_1 $w = 0$	Next state Y_2Y_1 $w = 1$	Output z
00	(00)	01	0
01	10	(01)	1
10	(10)	11	1
11	00	(11)	0

(a) Poor state assignment

Present state y_2y_1	Next state Y_2Y_1 $w = 0$	Next state Y_2Y_1 $w = 1$	Output z
00	(00)	01	0
01	11	(01)	1
11	(11)	10	1
10	00	(10)	0

(b) Good state assignment

Figure 9.14 State assignment for Figure 9.13*b*.

The uncertainty caused by multiple changes in the state variables in response to an input that should lead to a predictable change from one stable state to another has to be eliminated. The term *race condition* is used to refer to such unpredictable behavior. We will discuss this issue in detail in Section 9.5.

Race conditions can be eliminated by treating the present-state variables as if they were inputs to the circuit, meaning that only one state variable is allowed to change at a time. For our example the assignment $A = 00$, $B = 01$, $C = 11$, and $D = 10$ achieves this objective. The resulting excitation table is presented in Figure 9.14*b*. The reader should verify that all transitions involve changing a single state variable.

From Figure 9.14*b* the next-state and output expressions are

$$Y_1 = w\bar{y}_2 + \bar{w}y_1 + y_1\bar{y}_2$$
$$Y_2 = wy_2 + \bar{w}y_1 + y_1y_2$$
$$z = y_1$$

The last product term in the expressions for Y_1 and Y_2 is included to deal with possible hazards, which are discussed in Section 9.6. The corresponding circuit is shown in Figure 9.15.

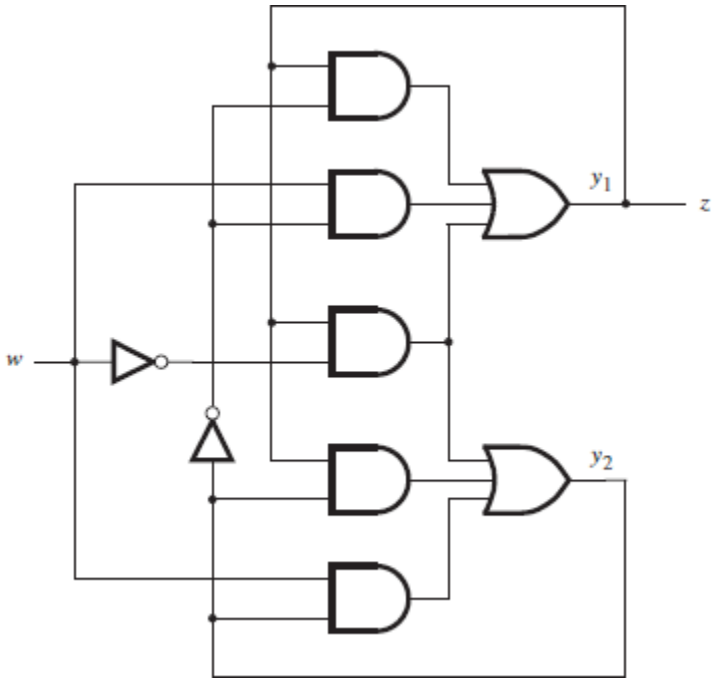


Figure 9.15 Circuit that implements the FSM in Figure 9.13*b*.

It is interesting to consider how the serial parity generator could be implemented using a synchronous approach. All that is needed is a single flip-flop that changes its state with the arrival of each input pulse. The positive-edge-triggered *D* flip-flop in Figure 9.16 accomplishes the task, assuming that the flip-flop is initially set to $Q = 0$. The logic complexity of the flip-flop is exactly the same as the circuit in Figure 9.15. Indeed, if we use the preceding expressions for Y_1 and Y_2 and substitute C for w, D for $\bar{y}_2$, y_m for y_1, and y_s for y_2, we end up with the excitation expressions shown for the M-S D flip-flop in Example 9.2. The circuit in Figure 9.15 is actually a negative-edge-triggered M-S flip-flop, with the complement of its Q output (y_2) connected to its *D* input. The output *z* is connected to the output of the M-stage of the flip-flop.

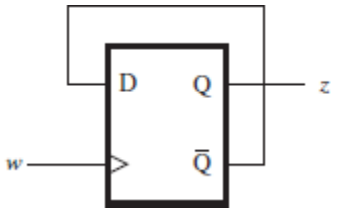


Figure 9.16 Synchronous solution for Example 9.4.

Example 9.5

MODULO-4 COUNTER Chapters 5 and 6 described how counters can be implemented using flip-flops. Now we will synthesize a counter as an asynchronous sequential circuit. Figure 9.17 depicts a state diagram for a modulo-4 up-counter, which counts the number of pulses on an input line, *w*. The circuit must be able to react to all changes in the input signal; thus it must take specific actions at both the positive and negative edges of each pulse. Therefore, eight states are needed to deal with the edges in four consecutive pulses.

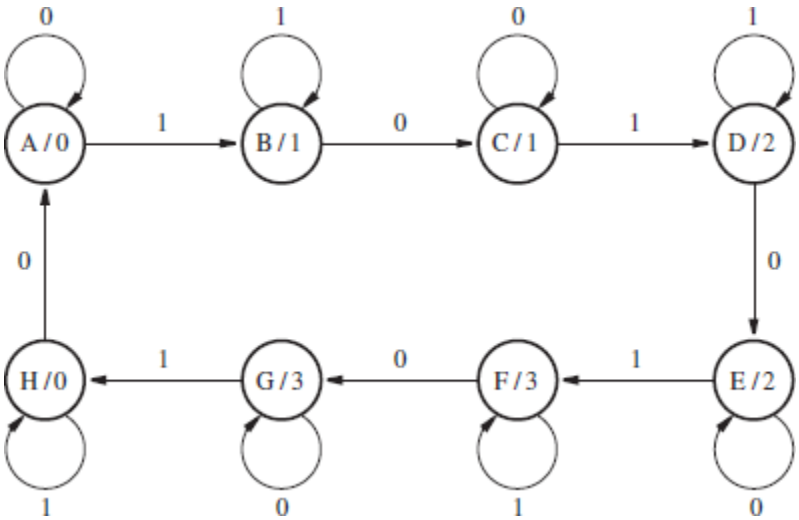


Figure 9.17 State diagram for a modulo-4 counter.

The counter begins in state *A* and stays in this state as long as $w = 0$. When *w* changes to 1, a transition to state *B* is made and the circuit remains stable in this state as long as $w = 1$. When *w* goes back to 0, the circuit moves to state *C* and remains stable until *w* becomes 1 again, which causes

a transition to state *D*, and so on. Using the Moore model, the states correspond to specific counts. There are two states for each particular count: the state that the FSM enters when *w* changes from 0 to 1 at the start of a pulse and the state that the FSM enters when *w* goes back to 0 at the end of the pulse. States *B* and *C* correspond to the count of 1, states *D* and *E* to 2, and states *F* and *G* to 3. States *A* and *H* represent the count of 0.

Figure 9.18 shows the flow and excitation tables for the counter. The state assignment is chosen such that all transitions between states require changing the value of only one state variable to eliminate the possibility of race conditions. In part *b* of the figure the output is encoded as a binary number, using variables z_2 and z_1. From the excitation table the next-state and output expressions are

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	(A)	B	0
B	C	(B)	1
C	(C)	D	1
D	E	(D)	2
E	(E)	F	2
F	G	(F)	3
G	(G)	H	3
H	A	(H)	0

(a) Flow table

Present state $y_3y_2y_1$	Next state $Y_3Y_2Y_1$		Output z_2z_1
	$w = 0$	$w = 1$	
000	(000)	001	00
001	011	(001)	01
011	(011)	010	01
010	110	(010)	10
110	(110)	111	10
111	101	(111)	11
101	(101)	100	11
100	000	(100)	00

(b) Excitation table

Mod-8 output $z_3z_2z_1$
000
001
010
011
100
101
110
111

(c) Output for counting the edges

Figure 9.18 Flow and excitation tables for a modulo-4 counter.

$$
\begin{aligned}
Y_1 &= \overline{w}y_1 + wy_2y_3 + w\overline{y}_2\overline{y}_3 + y_1y_2y_3 + y_1\overline{y}_2\overline{y}_3 \\
&= \overline{w}y_1 + (w + y_1)(y_2y_3 + \overline{y}_2\overline{y}_3) \\
Y_2 &= wy_2 + \overline{w}y_1\overline{y}_3 + \overline{y}_1y_2 + y_2\overline{y}_3 \\
Y_3 &= wy_3 + y_1y_3 + \overline{y}_1y_2\overline{w} + y_2y_3 \\
z_1 &= y_1 \\
z_2 &= y_1y_3 + \overline{y}_1y_2
\end{aligned}
$$

These expressions define the circuit that implements the required modulo-4 pulse counter.

In the preceding derivation we designed a circuit that changes its state on every edge of the input signal w, requiring a total of eight states. Since the circuit is supposed to count the number of complete pulses, which contain a rising and a falling edge, the output count z_2z_1 changes its value only in every second state. This FSM behaves like a synchronous sequential circuit in which the output count changes only as a result of w changing from 0 to 1.

Suppose now that we want to count the number of times the signal w changes its value, that is, the number of its edges. The state transitions specified in Figures 9.17 and 9.18 define an FSM that can operate as a modulo-8 counter for this purpose. We only need to specify a distinct output in each state, which can be done as shown in Figure 9.18*c*. The values of $z_3z_2z_1$ indicate the counting sequence $0, 1, 2, \ldots, 7, 0$. Using this specification of the output and the state assignment in Figure 9.18*b*, the resulting output expressions are

$$
\begin{aligned}
z_1 &= y_1 \oplus y_2 \oplus y_3 \\
z_2 &= y_2 \oplus y_3 \\
z_3 &= y_3
\end{aligned}
$$

Example 9.6

A SIMPLE ARBITER In computer systems it is often useful to have some resource shared by a number of different devices. Usually, the resource can be used by only one device at a time. When various devices need to use the resource, they have to request to do so. These requests are handled by an arbiter circuit. When there are two or more outstanding requests, the arbiter

may use some priority scheme to choose one of them, as already discussed in Section 6.8.

We will now consider an example of a simple arbiter implemented as an asynchronous sequential circuit. To keep the example small, suppose that two devices are competing for the shared resource, as indicated in Figure 9.19*a*. Each device communicates with the arbiter by means of two signals: *Request* and *Grant*. When a device needs to use the shared resource, it raises its request signal to 1. Then it waits until the arbiter responds with the grant signal.

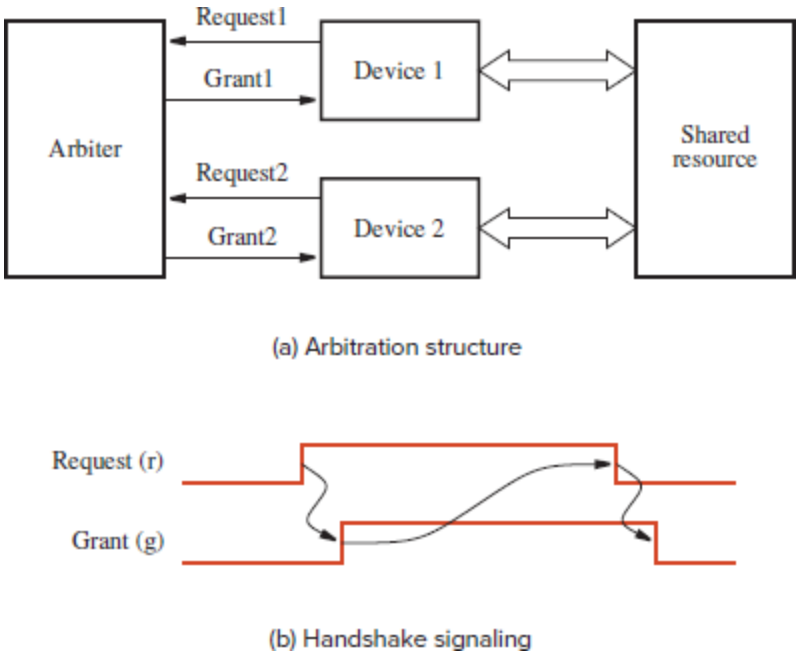


Figure 9.19 Arbitration example.

Figure 9.19*b* illustrates a commonly used scheme for communication between two entities in the asynchronous environment, known as *handshake signaling*. Two signals are used to provide the handshake. A device initiates the activity by raising a request, $r = 1$. When the shared resource is available, the arbiter responds by issuing a grant, $g = 1$. When the device receives the grant signal, it proceeds to use the requested shared resource. When it completes its use of the resource, it drops its request by setting $r = 0$. When the arbiter sees that $r = 0$, it deactivates the grant signal, making $g = 0$. The arrows in the figure indicate the cause-effect relationships in this signaling scheme; a change in one signal causes a change in the other signal. The time elapsed between the changes in the

cause-effect signals depends on the specific implementation of the circuit. A key point is that there is no need for a synchronizing clock.

A state diagram for our simple arbiter is given in Figure 9.20. There are two inputs, the request signals r_1 and r_2, and two outputs, the grant signals g_1 and g_2. The diagram depicts the Moore model of the required FSM, where the arcs are labeled as r_2r_1 and the state outputs as g_2g_1. The quiescent state is *A*, where there are no requests. State *B* represents the situation in which Device 1 is given permission to use the resource, and state *C* denotes the same for Device 2. Thus *B* is stable if $r_2\, r_1 = 01$, and *C* is stable if $r_2\, r_1 = 10$. To conform to the rules of asynchronous circuit design, we will assume that the inputs r_1 and r_2 become activated one at a time. Hence, in state *A* it is impossible to have a change from $r_2\, r_1 = 00$ to $r_2\, r_1 = 11$. The situation where $r_2\, r_1 = 11$ occurs only when a second request is raised before the device that has the grant signal completes its use of the shared resource, which can happen in states *B* and *C*. If the FSM is stable in either state *B* or *C*, it will remain in this state if both r_1 and r_2 go to 1.

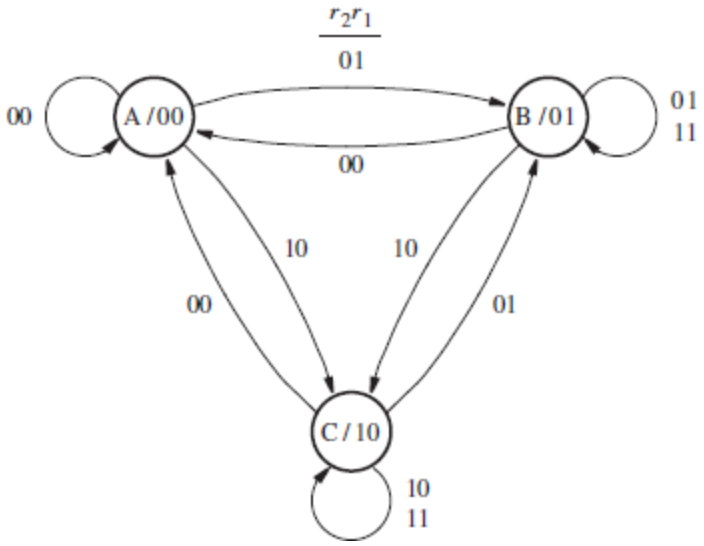


Figure 9.20 State diagram for the arbiter.

The flow table is given in Figure 9.21*a*, and the excitation table is presented in Figure 9.21*b*. It is impossible to choose a state assignment such that all changes between states *A*, *B*, and *C* involve a change in a single state variable only. In the chosen assignment the transitions to or from state *A* are handled properly, but the transitions between states *B* and *C* involve

changes in the values of both state variables y_1 and y_2. Suppose that the circuit is stable in state B under input valuation $r_2\ r_1 = 11$. Now let the inputs change to $r_2\ r_1 = 10$. This should cause a change to state C, which means that the state variables must change from $y_2\ y_1 = 01$ to 10. If y_1 changes faster than y_2, then the circuit will find itself momentarily in state $y_2\ y_1 = 00$, which leads to the desired final state because from state A there is a specified transition to C under the input valuation 10. But if y_2 changes faster than y_1, the circuit will reach the state $y_2\ y_1 = 11$, which is not defined in the flow table. To make sure that even in this case the circuit will proceed to the required destination C, we can include the state $y_2\ y_1 = 11$, labeled D, in the excitation table and specify the required transition as shown in the figure. A similar situation arises when the circuit is stable in C under $r_2\ r_1 = 11$, and it has to change to B when r_2 changes from 1 to 0.

Present state	Next state r_2r_1 = 00	01	10	11	Output g_2g_1
A	(A)	B	C	–	00
B	A	(B)	C	(B)	01
C	A	B	(C)	(C)	10

(a) Flow table

	Present state y_2y_1	Next state r_2r_1 = 00	01	10	11	Output g_2g_1
		Y_2Y_1				
A	00	(00)	01	10	–	00
B	01	00	(01)	10	(01)	01
C	10	00	01	(10)	(10)	10
D	11	–	01	10	–	*dd*

(b) Excitation table

Figure 9.21 Implementation of the arbiter.

The output values for the extra state D are indicated as don't cares. Whenever a specific output is changing from 0 to 1 or from 1 to 0, exactly

when this change takes place is not important if the correct value is produced when the circuit is in a stable state. The don't-care specification may lead to a simpler realization of the output functions. It is important to ensure that unspecified outputs will not result in a value that may cause erroneous behavior. From Figure 9.21*b* it is possible that during the short time when the circuit passes through the unstable state *D* the outputs become $g_2\, g_1 = 11$. This is harmless in our example because the device that has just finished using the shared resource will not try to use it again until its grant signal has returned to 0 to indicate the end of the handshake with the arbiter. Observe that if this condition occurs when changing from *B* to *C*, then g_1 remains 1 slightly longer and g_2 becomes 1 slightly earlier. Similarly, if the transition is from *C* to *B*, then the change in g_1 from 0 to 1 happens slightly earlier and g_2 changes to 0 slightly later. In both of these cases there is no glitch on either g_1 or g_2.

From the excitation table the following next-state and output expressions are derived

$$
\begin{aligned}
Y_1 &= \bar{r}_2 r_1 + r_1 \bar{y}_2 \\
Y_2 &= r_2 \bar{r}_1 + r_2 y_2 \\
g_1 &= y_1 \\
g_2 &= y_2
\end{aligned}
$$

Rewriting the first two expressions as

$$
\begin{aligned}
Y_1 &= r_1(\bar{r}_2 + \bar{y}_2) \\
&= r_1 \overline{r_2 y_2} \\
Y_2 &= r_2(\bar{r}_1 + y_2)
\end{aligned}
$$

produces the circuit in Figure 9.22. Observe that this circuit responds very quickly to the changes in the input signals. This behavior is in sharp contrast to the arbiter discussed in Section 6.8 in which the synchronizing clock determines the minimum response time.

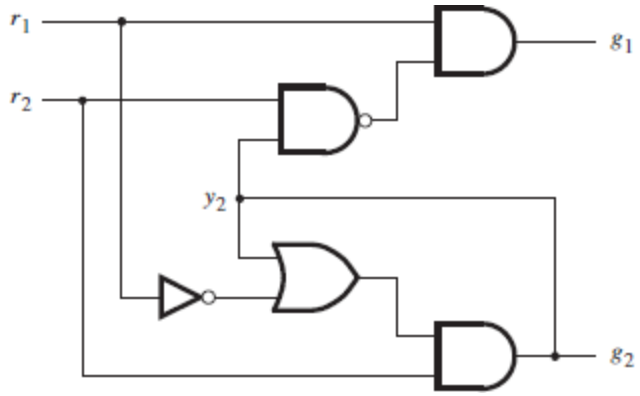


Figure 9.22 The arbiter circuit.

The difficulty with the race condition that arises in state changes between *B* and *C* can be resolved in another way. We can simply prevent the circuit from reaching an unspecified state. Figure 9.23*a* shows a modified flow table in which transitions between states *B* and *C* are made via state *A*. If the circuit is stable in *B* and the input valuation changes from $r_2\ r_1 = 11$ to 10, a change to *A* will occur first. As soon as the circuit reaches *A*, which is not stable for the input valuation 10, it will proceed to the stable state *C*. The detour through the unstable state *A* is acceptable because in this state the output is $g_2\ g_1 = 00$, which is consistent with the desired operation of the arbiter. The change from *C* to *B* is handled using the same approach. From the modified excitation table in Figure 9.23*b*, the following next-state expressions are derived

$$Y_1 = r_1\bar{y}_2$$
$$Y_2 = \bar{r}_1 r_2 \bar{y}_1 + r_2 y_2$$

Present state	Next state r_2r_1 = 00	01	10	11	Output g_2g_1
A	(A)	B	C	–	00
B	A	(B)	A	(B)	01
C	A	A	(C)	(C)	10

(a) Modified flow table

Present state y_2y_1	Next state r_2r_1 = 00	01	10	11	Output g_2g_1
	Y_2Y_1				
00	(00)	01	10	–	00
01	00	(01)	00	(01)	01
10	00	00	(10)	(10)	10

(b) Modified excitation table

Figure 9.23 An alternative for avoiding a critical race in Figure 9.21*a*.

These expressions give rise to a circuit different from the one in Figure 9.22. However, both circuits implement the functionality required in the arbiter.

Next we will attempt to design the same arbiter using the Mealy model specification. From Figure 9.20 it is apparent that the states *B* and *C* are fundamentally different because for the input $r_2\ r_1 = 11$ they must produce two different outputs. But state *A* is unique only to the extent that it generates the output $g_2\ g_1 = 00$ whenever $r_2\ r_1 = 00$. This condition could be specified in both *B* and *C* if the Mealy model is used. Figure 9.24 gives a suitable state diagram. The flow and excitation tables are presented in Figure 9.25, which lead to the following expressions

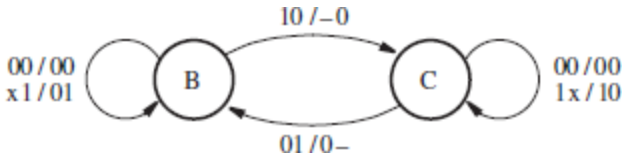


Figure 9.24 Mealy model for the arbiter FSM.

Present state	Next state				Output g_2g_1			
	r_2r_1 = 00	01	10	11	00	01	10	11
B	(B)	(B)	C	(B)	00	01	– 0	01
C	(C)	B	(C)	(C)	00	0–	10	10

(a) Flow diagram

Present state y	Next state				Output			
	r_2r_1 = 00	01	10	11	00	01	10	11
	Y				g_2g_1			
0	(0)	(0)	1	(0)	00	01	$d0$	01
1	(1)	0	(1)	(1)	00	$0d$	10	10

(b) Excitation table

Figure 9.25 Mealy model implementation of the arbiter FSM.

$$Y = r_2\bar{r}_1 + \bar{r}_1 y + r_2 y$$
$$g_1 = r_1\bar{y}$$
$$g_2 = r_2 y$$

Despite needing a single state variable, this circuit requires more gates for implementation than does the Moore version in Figure 9.22.

An important notion in the above examples is that it is necessary to pay careful attention to the state assignment, to avoid races in changing of the values of the state variables. Section 9.5 deals with this issue in more detail.

We made the basic assumption that the request inputs to the arbiter FSM change their values one at a time, which allows the circuit to reach a stable state before the next change takes place. If the devices are totally independent, they can raise their requests at any time. Suppose that each device raises a request every few seconds. Since the arbiter circuit needs only a few nanoseconds to change from one stable state to another, it is quite unlikely that both devices will raise their requests so close to each other that the arbiter circuit will produce erroneous outputs. However, while the probability of an error caused by the simultaneous arrival of requests is extremely low, it is not zero. If this small possibility of an error cannot be tolerated, then it is possible to feed the request signals through a special circuit called the *mutual exclusion* (ME) element. This circuit has two

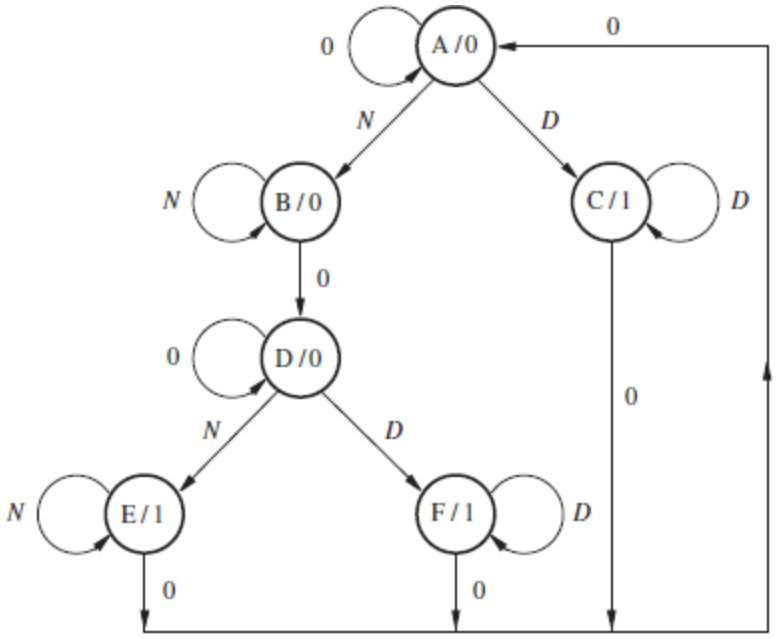
inputs and two outputs. If both inputs are 0, then both outputs are 0. If only one input is 1, then the corresponding output is 1. If both inputs are 1, the circuit makes one output go to 1 and keeps the other at 0. Using the ME element would change the design of the arbiter slightly; because the valuation $r_2\ r_1 = 11$ would never occur, all entries in the corresponding column in Figure 9.21 would be don't cares. The ME element and the issue of simultaneous changes in input signals are discussed in detail in reference. Finally, we should note that a similar problem arises in synchronous circuits in which one or more inputs are generated by a circuit that is not controlled by a common clock, as discussed in Section 7.8.3.

9.4 State Reduction

In Chapter 6 we saw that reducing the number of states needed to realize the functionality of a given FSM usually leads to fewer state variables, which means that fewer flip-flops are required in the corresponding synchronous sequential circuit. In asynchronous sequential circuits it is also useful to try to reduce the number of states because this usually results in simpler implementations.

When designing an asynchronous FSM, the initial flow table is likely to have many unspecified (don't-care) entries, because the designer has to obey the restriction that only one input variable can change its value at a time. For example, suppose that we want to design the FSM for the simple vending machine considered in Example 9.3. Recall that the machine accepts nickels and dimes and dispenses candy when 10 cents is deposited; the machine does not give change if 15 cents is deposited. An initial state diagram for this FSM can be derived in straightforward fashion by enumerating all possible sequences of depositing the coins to give a sum of at least 10 cents. Figure 9.26*a* shows a possible diagram, defined as a Moore model. Starting in a reset state, *A*, the FSM remains in this state as long as no coin is deposited. This is denoted by an arc labeled 0 to indicate that $N = D = 0$. Now let an arc with the label *N* denote that the coin-sensing mechanism has detected a nickel and has generated a signal $N = 1$. Similarly, let *D* denote that a dime has been deposited. If $N = 1$, then the FSM has to move to a new state, say, *B*, and it must remain stable in this

state as long as N has the value of 1. Since B corresponds to 5 cents being deposited, the output in this state has to be 0. If a dime is deposited in state A, then the FSM must move to a different state, say, C. The machine should stay in C as long as $D = 1$, and it should release the candy by generating the output of 1. These are the only possible transitions from state A, because it is impossible to insert two coins at the same time, which means that $DN = 11$ can be treated as a don't-care condition. Next, in state B there must be a return to the condition $DN = 00$ because the coin-sensing mechanism will detect the second coin some time after the first coin has cleared the mechanism. This behavior is consistent with the requirement that only one input variable can change at a time; hence it is not allowed to go from $DN = 01$ to $DN = 10$. The input $DN = 10$ cannot occur in state B and should be treated as a don't care. The input $DN = 00$ takes the FSM to a new state, D, which indicates that 5 cents has been deposited and that there is no coin in the sensing mechanism. In state D it is possible to deposit either a nickel or a dime. If $DN = 01$, the machine moves to state E, which denotes that 10 cents has been deposited and generates the output of 1. If $DN = 10$, the machine moves to state F, which also generates the output of 1. Finally, when the FSM is in any of the states C, E, or F, the only possible input is $DN = 00$, which returns the machine to state A.



(a) Initial state diagram

Present state	Next state DN = 00	01	10	11	Output z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	A	–	(C)	–	1
D	(D)	E	F	–	0
E	A	(E)	–	–	1
F	A	–	(F)	–	1

(b) Initial flow table

Figure 9.26 Derivation of an FSM for the simple vending machine.

The flow table for this FSM is given in Figure 9.26*b*. It shows explicitly all don't-care entries. Such unspecified entries provide a certain amount of flexibility that can be exploited in reducing the number of states. Note that in each row of this table there is only one stable state. Such tables, where there is only one stable state for each row, are often referred to as *primitive flow tables*.

Several techniques have been developed for state reduction. In this section we will describe a two-step process. In the first step we will apply the partitioning procedure from Section 6.6.1, assuming that the potentially

equivalent rows in a flow table must produce the same outputs. As an additional constraint, for two rows to be potentially equivalent any unspecified entries must be in the same next-state columns. Thus combining the equivalent states into a single state will not remove the don't cares and the flexibility that they provide. In the second step, the rows are *merged* exploiting the unspecified entries. Two rows can be merged if they have no conflicting next-state entries. This means that their next-state entries for any given valuation of inputs are either the same, or one of them is unspecified, or both rows indicate a stable state. If the Moore model is used, then the two rows (states) must produce the same outputs. If the Mealy model is used, then the two states must produce the same outputs for any input valuations for which both states are stable.

Example 9.7

we will now show how the flow diagram in Figure 9.26*b* can be reduced to the optimized form in Figure 9.12. The first step in the state-reduction process is the partitioning procedure from Section 6.6.1. States A and D are stable under the input valuation $DN = 00$, producing the output of 0; they also have the unspecified entries in the same position. States C and F are stable under $DN = 10$, generating $z = 1$, and they have the same unspecified entries. States B and E have the same unspecified entries, but when they are stable under $DN = 01$ the state B produces $z = 0$ while E generates $z = 1$; they are not equivalent. Therefore, the initial partition is

$$P_1 = (AD)(B)(CF)(E)$$

The successors of A and D are (A,D) for $DN = 00$, (B,E) for 01, and (C,F) for 10. Since the (B,E) pair is not in the same block of P_1, it follows that A and D are not equivalent. The successors of C and F are (A,A) for 00 and (C,F) for 10; each pair is in a single block. Thus the second partition is

$$P_2 = (A)(D)(B)(CF)(E)$$

The successors of C and F in P_2 are in the same block of P_2, which means that

$$P_3 = P_2$$

The conclusion is that rows C and F are equivalent. Combining them into a single row and changing all Fs into Cs gives the flow table in Figure 9.27.

Present state	Next state				Output
	DN = 00	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	A	–	(C)	–	1
D	(D)	E	C	–	0
E	A	(E)	–	–	1

Figure 9.27 First-step reduction of the FSM in Figure 9.26*b*.

Next we can try to merge some rows in the flow table by exploiting the existence of unspecified entries. The only row that can be merged with others is C. It can be merged with either A or E, but not both. Merging C with A would mean that the new state has to generate $z = 0$ when it is stable under the input valuation 00 and has to produce $z = 1$ when stable under 10. This can be achieved only by using the Mealy model. The alternative is to merge C and E, in which case the new state is stable under DN = 01 and 10, producing the output of 1. This can be achieved with the Moore model. Merging C and E into a single state C and changing all Es into Cs yields the reduced flow table in Figure 9.12. Observe that when C and E are merged, the new row C must include all specifications in both rows C and E. Both rows specify A as the next state if DN = 00. Row E specifies a stable state for DN = 01; hence the new row (called C) must also specify a stable state for the same valuation. Similarly, row C specifies a stable state for DN = 10, which must be reflected in the new row. Therefore, the next-state entries in the new row are A, (C) and (C) for the input valuations 00, 01, and 10, respectively.

Merging Procedure

In Example 9.7 it was easy to decide which rows should be merged because the only possibilities are to merge row C with either A or E. We chose to merge C and E because this can be done preserving the Moore

model, which is likely to lead to a simpler expression that realizes the output z.

In general, there can be many possibilities for merging rows in larger flow tables. In such cases it is necessary to have a more structured procedure for making the choice. A useful procedure can be defined using the concept of *compatibility* of states.

Definition 9.1 – *Two states (rows in a flow table), S_i and S_j, are said to be compatible if there are no state conflicts for any input valuation. Thus for each input valuation, one of the following conditions must be true:*

- *both S_i and S_j have the same successor, or*
- *both S_i and S_j are stable, or*
- *the successor of S_i or S_j, or both, is unspecified.*

Moreover, both S_i and S_j must have the same output whenever specified.

Consider the primitive flow table in Figure 9.28. Let us examine the compatibility between different states, assuming that we would like to retain the Moore-type specification of outputs for this FSM. State *A* is compatible only with state *H*. State *B* is compatible with states *F* and *G*. State *C* is not compatible with any other state. State *D* is compatible with state *E*; so are state *F* with *G* and state *G* with *H*. In other words, the following compatible pairs exist: (*A*,*H*), (*B*,*F*), (*B*,*G*), (*D*,*E*), (*F*,*G*), and (*G*,*H*). The compatibility relationship among various states can be represented conveniently in the form of a *merger diagram*, as follows:

Present state	Next state				Output z
	$w_2w_1 =$ 00	01	10	11	
A	(A)	H	B	–	0
B	F	–	(B)	C	0
C	–	H	–	(C)	1
D	A	(D)	–	E	1
E	–	D	G	(E)	1
F	(F)	D	–	–	0
G	F	–	(G)	–	0
H	–	(H)	–	E	0

Figure 9.28 A primitive flow table.

- Each row of the flow table is represented as a point, labeled by the name of the row.
- A line is drawn connecting any two points that correspond to compatible states (rows).

From the merger diagram the best merging possibility can be chosen, and the reduced flow table can be derived.

Figure 9.29 gives the merger diagram for the primitive flow table in Figure 9.28. The diagram indicates that row *A* can be merged with *H*, but only if *H* is not merged with *G*, because there is no line joining *A* and *G*. Row *B* can be merged with rows *F* and *G*. Since it is also possible to merge *F* and *G*, it follows that *B*, *F*, and *G* are pairwise compatible. Any set of rows that are pairwise compatible for all pairs in the set can be merged into a single state. Thus states *B*, *F*, and *G* can be merged into a single state, but only if states *G* and *H* are not merged. State *C* cannot be merged with any other state. States *D* and *E* can be merged.

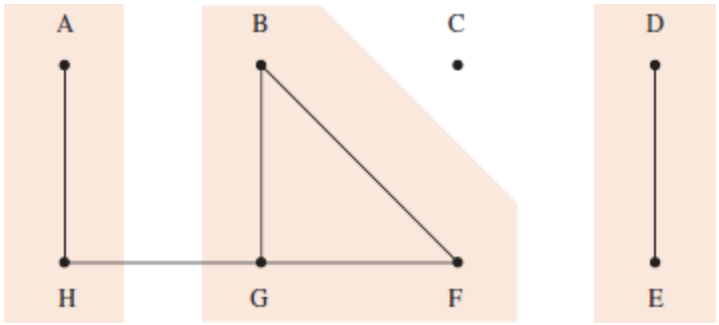


Figure 9.29 Merger diagram for the flow table in Figure 9.28, which preserves the Moore model.

A prudent strategy is to merge the states so that the resulting flow table has as few states as possible. In our example the best choice is to merge the compatibles (*A*,*H*), (*B*,*F*,*G*), and (*D*,*E*), which leads to the reduced flow table in Figure 9.30. When a new row is created by merging two or more rows, all entries in the new row have to be specified to cover the individual requirements of the constituent rows. Replacing rows *A* and *H* with a new row *A* requires making *A* stable for both $w_2\ w_1$ = 00 and 01, because the old *A* has to be stable for 00 and *H* has to be stable for 01. It also requires

specifying B as the next-state for $w_2\ w_1 = 10$ and E as the next state for $w_2\ w_1 = 11$. Since the old state E becomes D, after merging D and E, the new row A must have the next-state entries (A), (A), B, and D for the input valuations 00, 01, 10, and 11, respectively. Replacing rows B, F, and G with a new row B requires making B stable for $w_2\ w_1 = 00$ and 10. The next-state entry for $w_2\ w_1 = 01$ has to be D to satisfy the requirement of the old state F. The next-state entry for $w_2\ w_1 = 11$ has to be C, as dictated by the old state B. Observe that the old state G imposes no requirements for transitions under $w_2\ w_1 = 01$ and 11, because its corresponding next-state entries are unspecified. Row C remains the same as before except that the name of the next-state entry for $w_2\ w_1 = 01$ has to be changed from H to A. Rows D and E are replaced by a new row D, using similar reasoning. Note that the flow table in Figure 9.30 is still of Moore type.

Present state	Next state				Output
	$w_2w_1 =$ 00	01	10	11	z
A	(A)	(A)	B	D	0
B	(B)	D	(B)	C	0
C	–	A	–	(C)	1
D	A	(D)	B	(D)	1

Figure 9.30 Reduced Moore-type flow table for the FSM in Figure 9.28.

So far we considered merging only those rows that would allow us to retain the Moore-type specification of the FSM in Figure 9.28. If we are willing to change to the Mealy model, then other possibilities exist for merging. Figure 9.31 shows the complete merger diagram for the FSM of Figure 9.28. Black lines connect the compatible states that can be merged into a new state that has a Moore-type output; this corresponds to the merger diagram in Figure 9.29. Red lines connect the states that can be merged only if Mealy-type outputs are used.

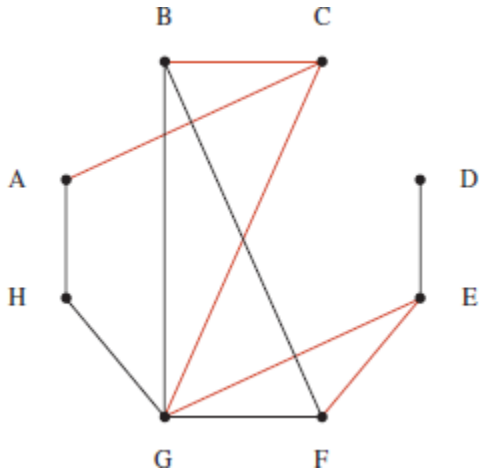


Figure 9.31 Complete merger diagram for Figure 9.28.

In this case going to the Mealy model is unlikely to result in a simpler circuit. Although several merger possibilities exist, they all require at least four states in the reduced flow table, which is not any better than the solution obtained in Figure 9.30. For example, one possibility is to perform the merge based on the partition (*A*,*H*), (*B*,*C*,*G*) (*D*,*E*) (*F*). Another possibility is to use (*A*,*C*) (*B*,*F*) (*D*,*E*) (*G*,*H*). We will not pursue these possibilities and will discuss the issues involved in specifying the Mealy-type outputs in Example 9.9.

State Reduction Procedure

We can summarize the steps needed to generate the reduced flow table from a primitive flow table as follows:

1. Use the partitioning procedure to eliminate the equivalent states in a primitive flow table.
2. Construct a merger diagram for the resulting flow table.
3. Choose subsets of compatible states that can be merged, trying to minimize the number of subsets needed to cover all states. Each state must be included in only one of the chosen subsets.
4. Derive the reduced flow table by merging the rows in chosen subsets.
5. Repeat steps 2 to 4 to see whether further reductions are possible.

Choosing an optimal subset of compatible states for merging can be a very complicated task because for large FSMs there may be many possibilities that should be investigated. A trial-and-error approach is a reasonable way to tackle this problem.

Example 9.8

Consider the initial flow table in Figure 9.32. To apply the partitioning procedure, we identify state pairs (*A*,*G*), (*B*,*L*), and (*H*,*K*) as being potentially equivalent rows, because both rows in each pair have the same outputs and their don't-care entries are in the same column. The remaining rows are distinct in this respect. Therefore, the first partition is

$$P_1 = (AG)(BL)(C)(D)(E)(F)(HK)(J)$$

Present state	Next state				Output
	w_2w_1 = 00	01	10	11	z
A	(A)	F	C	–	0
B	A	(B)	–	H	1
C	G	–	(C)	D	0
D	–	F	–	(D)	1
E	G	–	(E)	D	1
F	–	(F)	–	K	0
G	(G)	B	J	–	0
H	–	L	E	(H)	1
J	G	–	(J)	–	0
K	–	B	E	(K)	1
L	A	(L)	–	K	1

Figure 9.32 Flow table for Example 9.8.

Now the successors of (*A*,*G*) are (*A*,*G*) for $w_2\ w_1$ = 00, (*F*,*B*) for 01, and (*C*,*J*) for 10. Since *F* and *B*, as well as *C* and *J*, are not in the same block, it follows that *A* and *G* are not equivalent. The successors of (*B*,*L*) are (*A*,*A*), (*B*,*L*), and (*H*,*K*), respectively. All are in single blocks. The successors of (*H*,*K*) are (*L*,*B*), (*E*,*E*), and (*H*,*K*), which are all contained in single blocks. Therefore, the second partition is

$$P_2 = (A)(G)(BL)(C)(D)(E)(F)(HK)(J)$$

Repeating the successor test shows that the successors of (*B*,*L*) and (*H*,*K*) are still in single blocks; hence

$$P_3 = P_2$$

Combining rows *B* and *L* under the name *B* and rows *H* and *K* under the name *H* leads to the flow table in Figure 9.33.

Present state	Next state				Output
	w_2w_1 = 00	01	10	11	z
A	(A)	F	C	–	0
B	A	(B)	–	H	1
C	G	–	(C)	D	0
D	–	F	–	(D)	1
E	G	–	(E)	D	1
F	–	(F)	–	H	0
G	(G)	B	J	–	0
H	–	B	E	(H)	1
J	G	–	(J)	–	0

Figure 9.33 Reduction obtained by using the partitioning procedure.

A merger diagram for this flow table is given in Figure 9.34. It indicates that rows *B* and *H* should be merged into one row, which we will label as *B*. The merger diagram also suggests that rows *D* and *E* should be merged; we will call the new row *D*. The remaining rows present more than one choice for merging. Rows *A* and *F* can be merged, but in that case *F* and *J* cannot be merged. Rows *C* and *J* can be merged, or *G* and *J* can be merged. We will choose to merge the rows *A* and *F* into a new row called *A* and rows *G* and *J* into a new row *G*. The merger choice is indicated in red in the diagram. The resultant flow table is shown in Figure 9.35. To see whether this table offers any further opportunities for merging, we can construct the merger diagram in Figure 9.36. From this diagram it is apparent that rows *C* and *G* can be merged; let the new row be called *C*. This leads to the flow table in Figure 9.37, which cannot be reduced any more.

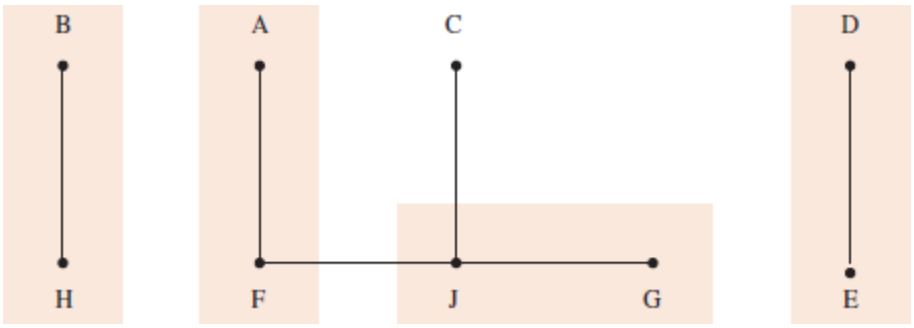


Figure 9.34 Merger diagram for Figure 9.33.

Present state	Next state w_2w_1 = 00	01	10	11	Output z
A	(A)	(A)	C	B	0
B	A	(B)	D	(B)	1
C	G	–	(C)	D	0
D	G	A	(D)	(D)	1
G	(G)	B	(G)	–	0

Figure 9.35 Reduction obtained from the merger diagram in Figure 9.34.

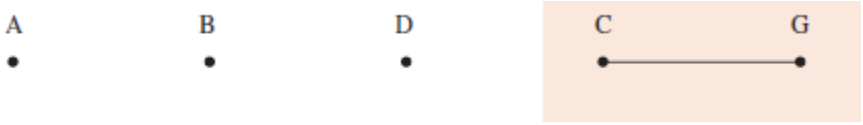


Figure 9.36 Merger diagram for Figure 9.35.

Present state	Next state w_2w_1 = 00	01	10	11	Output z
A	(A)	(A)	C	B	0
B	A	(B)	D	(B)	1
C	(C)	B	(C)	D	0
D	C	A	(D)	(D)	1

Figure 9.37 Reduced flow table for Example 9.8.

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	F	C	–	0
B	A	(B)	–	H	1
C	G	–	(C)	D	0
D	–	F	–	(D)	1
E	G	–	(E)	D	1
F	–	(F)	–	K	0
G	(G)	B	J	–	0
H	–	L	E	(H)	1
J	G	–	(J)	–	0
K	–	B	E	(K)	1
L	A	(L)	–	K	1

Figure 9.32 Flow table for Example 9.8.

Example 9.9

Consider the flow table in Figure 9.38. Applying the partitioning procedure to this table gives

$$P_1 = (AFK)(BJ)(CG)(D)(E)(H)$$
$$P_2 = (A)(FK)(BJ)(C)(G)(D)(E)(H)$$
$$P_3 = P_2$$

Present state	Next state				Output
	w_2w_1 = 00	01	10	11	z
A	(A)	G	E	–	0
B	K	–	(B)	D	0
C	F	(C)	–	H	1
D	–	C	E	(D)	0
E	A	–	(E)	D	1
F	(F)	C	J	–	0
G	K	(G)	–	D	1
H	–	–	E	(H)	1
J	F	–	(J)	D	0
K	(K)	C	B	–	0

Figure 9.38 Flow table for Example 9.9.

Combining *B* and *J* into a new state *B*, and *F* and *K* into *F*, gives the flow table in Figure 9.39.

Present state	Next state				Output
	w_2w_1 = 00	01	10	11	z
A	(A)	G	E	–	0
B	F	–	(B)	D	0
C	F	(C)	–	H	1
D	–	C	E	(D)	0
E	A	–	(E)	D	1
F	(F)	C	B	–	0
G	F	(G)	–	D	1
H	–	–	E	(H)	1

Figure 9.39 Reduction resulting from the partitioning procedure.

Figure 9.40*a* gives a merger diagram for this flow table, indicating the possibilities for merger if the Moore model of the FSM is to be preserved. In this case *B* and *F* can be merged, as well as *C* and *H*, resulting in a six-row flow table.

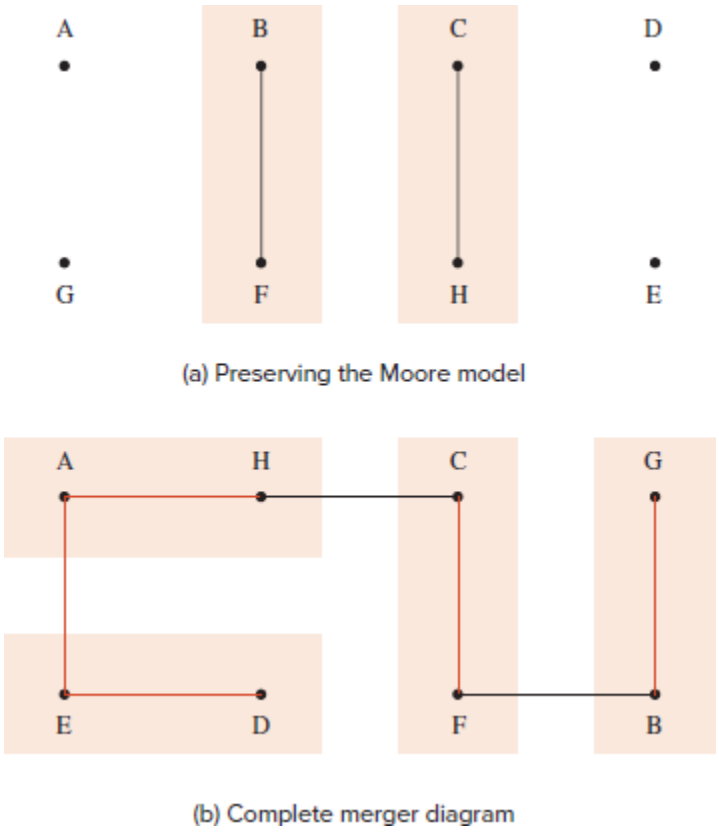


Figure 9.40 Merger diagram for Figure 9.39.

Next we should consider the merging possibilities if we are willing to change to the Mealy model. When going from the Moore model to the Mealy model, a stable state in the Mealy model must generate the same output as it had in the Moore model. It is also important to ensure that transitions in the Mealy model will not produce undesirable glitches in the output signal.

Figure 9.41 indicates how the FSM of Figure 9.39 can be represented in the Mealy form. The next-state entries are unchanged. In Figure 9.41, for each stable state the output value must be the same as for the corresponding row of the Moore-type table. For example, $z = 0$ when the state A is stable under $w_2\ w_1 = 00$. Also, $z = 0$ when the states B, D, and F are stable under $w_2\ w_1 = 10$, 11, and 00, respectively. Similarly, $z = 1$ when C, E, G, and H are stable under $w_2\ w_1 = 01$, 10, 01, and 11, respectively. If a transition from one stable state to another requires the output to change from 0 to 1, or from 1 to 0, then the exact time when the change takes place is not important, as we explained in Section 9.1 when discussing Figure 9.3. For instance,

suppose that the FSM is stable in A under $w_2\ w_1 = 00$, producing $z = 0$. If the inputs then change to $w_2\ w_1 = 01$, a transition to state G must be made, where $z = 1$. Since it is not essential that z becomes 1 before the circuit reaches the state G, the output entry in row A that corresponds to this transition can be treated as a don't care; therefore, it is left unspecified in the table. From the stable state A, it is also possible to change to E, which allows specifying another don't care because z changes from 0 to 1. A different situation arises in row B. Suppose that the circuit is stable in B under $w_2\ w_1 = 10$ and that the inputs change to 11. This has to cause a change to stable state D, and z must remain at 0 throughout the change in states. Hence the output in row B under $w_2\ w_1 = 11$ is specified as 0. If it were left unspecified, to be used as a don't care, then it is possible that in the implementation of the circuit this don't care may be treated as a 1. This would cause a glitch in z, which would change $0 \rightarrow 1 \rightarrow 0$ as the circuit moves from B to D when the inputs change from 10 to 11. The same situation occurs for the transition from B to F when the inputs change from 10 to 00. We can use the same reasoning to determine other output entries in Figure 9.41.

Present state	Next state				Output z			
	$w_2w_1 =$ 00	01	10	11	00	01	10	11
A	(A)	G	E	–	0	–	–	–
B	F	–	(B)	D	0	–	0	0
C	F	(C)	–	H	–	1	–	1
D	–	C	E	(D)	–	–	–	0
E	A	–	(E)	D	–	–	1	–
F	(F)	C	B	–	0	–	0	–
G	F	(G)	–	D	–	1	–	–
H	–	–	E	(H)	–	–	1	1

Figure 9.41 The FSM of Figure 9.39 specified in the form of the Mealy model.

From Figure 9.41 we can derive the merger diagram in Figure 9.40*b*. The red lines connect the rows that can be merged only by specifying the output in the Mealy style. The black lines connect the rows that can be merged even if the outputs are of Moore type; they correspond to the diagram in

Figure 9.40*a*. Choosing the subsets of compatible states (*A*, *H*), (*B*, *G*), (*C*, *F*), and (*D*, *E*), the FSM can be represented using only four states. Merging the states *A* and *H* into a new state *A*, states *B* and *G* into *B*, states *C* and *F* into *C*, and *D* and *E* into *D*, we obtain the reduced flow table in Figure 9.42. Each entry in this table meets the requirements specified in the corresponding rows that were merged.

Present state	Next state				Output z			
	w_2w_1 = 00	01	10	11	00	01	10	11
A	(A)	B	D	(A)	0	–	1	1
B	C	(B)	(B)	D	0	1	0	0
C	(C)	(C)	B	A	0	1	0	1
D	A	C	(D)	(D)	–	–	1	0

Figure 9.42 Reduced flow table for Example 9.9.

Example 9.10

As another example consider the flow table in Figure 9.43. The partitioning procedure gives

Present state	Next state				Output z
	w_2w_1 = 00	01	10	11	
A	(A)	B	C	–	0
B	F	(B)	–	H	0
C	F	–	(C)	H	0
D	(D)	G	C	–	1
E	A	(E)	–	H	0
F	(F)	E	C	–	0
G	D	(G)	–	H	0
H	–	G	C	(H)	1

Figure 9.43 Flow table for Example 9.10.

$$P_1 = (AF)(BEG)(C)(D)(H)$$
$$P_2 = (AF)(BE)(G)(C)(D)(H)$$
$$P_3 = P_2$$

Replacing state F with A, and state E with B, results in the flow table in Figure 9.44. The corresponding merger diagram is presented in Figure 9.45. It is apparent that states A, B, and C can be merged and replaced with a new state A. Also D, G, and H can be merged into a new state D. The result is the reduced flow table in Figure 9.46, which has only two rows. Again we have used the Mealy model because the merged stable states D and H have $z = 1$ while G has $z = 0$.

Present state	Next state w_2w_1 = 00	01	10	11	Output z
A	(A)	B	C	–	0
B	A	(B)	–	H	0
C	A	–	(C)	H	0
D	(D)	G	C	–	1
G	D	(G)	–	H	0
H	–	G	C	(H)	1

Figure 9.44 Reduction after the partitioning procedure.

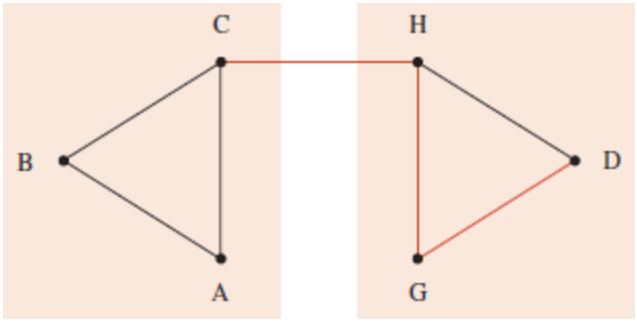


Figure 9.45 Merger diagram for Figure 9.44.

Present state	Next state w_2w_1 = 00	01	10	11	Output z 00	01	10	11
A	(A)	(A)	(A)	D	0	0	0	–
D	(D)	(D)	A	(D)	1	0	–	1

Figure 9.46 Reduced flow diagram for Example 9.10.

9.5 State Assignment

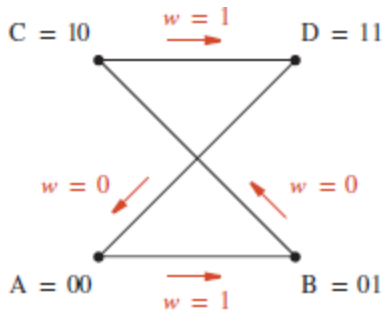
The examples in Section 9.3 illustrate that the state assignment task for asynchronous FSMs is complex. The time needed to change the value of a state variable depends on the propagation delays in the circuit. Thus it is impossible to ensure that a change in the values of two or more variables will take place at exactly the same time. To achieve reliable operation of the circuit, the state variables should change their values one at a time in controlled fashion. This is accomplished by designing the circuit such that a change from one state to another entails a change in one state variable only.

States in FSMs are encoded as bit strings that represent different valuations of the state variables. The number of bit positions in which two given bit strings differ is called the *Hamming distance* between the strings. For example, for bit strings 0110 and 0100 the Hamming distance is 1, while for 0110 and 1101 it is 3. Using this terminology, an ideal state assignment has a Hamming distance of 1 for all transitions from one stable state to another. When the ideal state assignment is not possible, an alternative that makes use of unspecified states and/or transitions through unstable states must be sought. Sometimes it is necessary to increase the number of state variables to provide the needed flexibility.

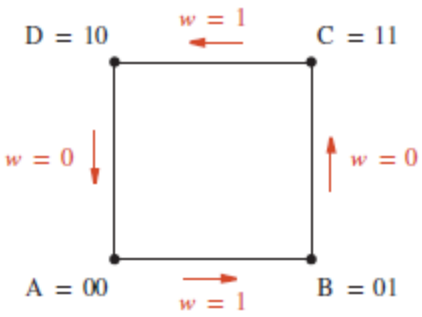
Example 9.11

Consider the parity-generating FSM in Figure 9.13. Two possible state assignments for this FSM are given in Figure 9.14. The transitions between states, as specified in Figure 9.13*b*, can be described in pictorial form as shown in Figure 9.47. Each row of the flow table is represented by a point. The four points needed to represent the rows are placed as vertices of a square. Each vertex has an associated code that represents a valuation of the state variables, y_2y_1. The codes shown in the figure, with $y_2y_1 = 00$ in the lower-left corner and so on, correspond to the coordinates of the two-dimensional cube presented in Section 8.3.1. Figure 9.47*a* shows what happens if the state assignment in Figure 9.14*a* is used; namely, if $A = 00$, $B = 01$, $C = 10$, and $D = 11$. There is a transition from A to B if $w = 1$, which requires a change in y_1 only. A transition from C to D occurs if $w = 1$, which also requires a change in y_1 only. However, a transition from B to C

caused by $w = 0$ involves a change in the values of both y_2 and y_1. Similarly, both state variables must change in going from D to A if $w = 0$. A change in both variables corresponds to a diagonal path in the diagram.



(a) Corresponding to Figure 9.14*a*



(b) Corresponding to Figure 9.14*b*

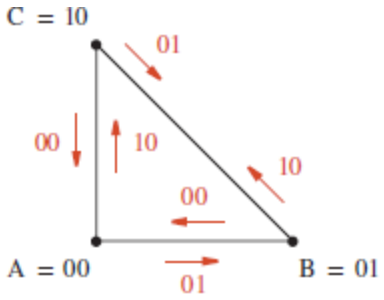
Figure 9.47 Transitions in Figure 9.13.

Figure 9.47*b* shows the effect of the state assignment in Figure 9.14*b*, which reverses the valuations assigned to C and D. In this case all four transitions are along the edges of the two-dimensional cube, and they involve a change in only one of the state variables. This is the desirable state assignment.

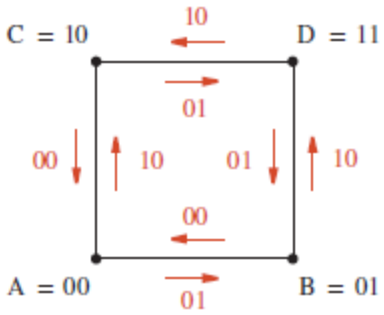
Example 9.12

The flow table for an arbiter FSM is given in Figure 9.21*a*. Transitions for this FSM are shown in Figure 9.48*a*, using the state assignment $A = 00$, $B = 01$, and $C = 10$. In this case multiple transitions are possible between the states. For example, there are two transitions between A and B: from B to A

if $r_2\ r_1 = 00$ and from A to B if $r_2\ r_1 = 01$. Again there is a diagonal path, corresponding to transitions between B and C, which should be avoided. A possible solution is to introduce a fourth state, D, as indicated in Figure 9.48*b*. Now the transitions between B and C can take place via the unstable state D. Thus instead of going directly from B to C when $r_2\ r_1 = 10$, the circuit will go first from B to D and then from D to C.



(a) Transitions in Figure 9.21*a*



(b) Using the extra state *D*

Figure 9.48 Transitions for the arbiter FSM in Figure 9.21.

Using the arrangement in Figure 9.48*b* requires modifying the flow table as shown in Figure 9.49. The state D is not stable for any input valuation. It cannot be reached if $r_2\ r_1 = 00$ or 11; hence these entries are left unspecified in the table. Also observe that we have specified the output $g_2\ g_1 = 10$ for state D, rather than leaving it unspecified. When a transition from one stable state to another takes place via an unstable state, the output of the unstable state must be the same as the output of one of the two stable states involved in the transition to ensure that a wrong output is not generated while passing through the unstable state.

Present state	Next state				Output
	r_2r_1 = 00	01	10	11	g_2g_1
A	(A)	B	C	–	00
B	A	(B)	D	(B)	01
C	A	D	(C)	(C)	10
D	–	B	C	–	10

Figure 9.49 Modified flow table based on the transitions in Figure 9.48*b*.

It is interesting to compare this flow table with the excitation table in Figure 9.21*b*, which is also based on using the extra state *D*. In Figure 9.21*b* the state *D* specifies the necessary transitions should the circuit accidentally find itself in this state as a result of a race in changing the values of both state variables. In Figure 9.49 the state *D* is used in orderly transitions, which are not susceptible to any race conditions.

9.5.1 TRANSITION DIAGRAM

A diagram that illustrates the transitions specified in a flow table is called a *transition diagram*. In some books such diagrams are called *state-adjacency diagrams*. These diagrams provide a convenient aid in searching for a suitable state assignment.

A good state assignment results if the transition diagram does not have any diagonal paths. A general way of stating this requirement is to say that it must be possible to *embed* the transition diagram onto a *k*-dimensional cube, because in a cube all transitions between adjacent vertices involve the Hamming distance of 1. Ideally, a transition diagram for an FSM with *n* state variables can be embedded onto an *n*-dimensional cube, as is the case in the examples in Figures 9.47*b* and 9.48*b*. If this is not possible, then it becomes necessary to introduce additional state variables, as we will see in later examples.

The diagrams in Figures 9.47 and 9.48 present all information pertinent to transitions between the states in the given FSMs. For larger FSMs such diagrams take on a cluttered appearance. A simpler form can be used instead, as described below.

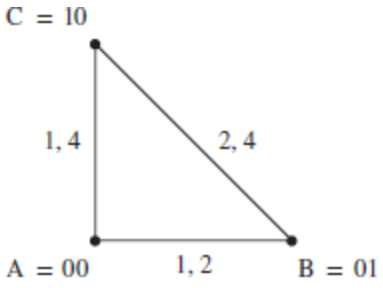
A transition diagram has to show the state transitions for each valuation of the input variables. The direction of a transition, for example from *A* to *B*

or from *B* to *A*, is not important, because it is only necessary to ensure that all transitions involve the Hamming distance of 1. The transition diagram has to show the effect of individual transitions into each stable state, which may involve passing through unstable states. For a given row of a flow table, it is possible to have two or more stable-state entries for different input valuations. It is useful to identify the transitions leading into these stable states with distinct labels in a transition diagram. To give each stable-state entry a distinct label, we will denote the stable-state entries with numbers 1, 2, 3, … . Thus if state *A* is stable for two input valuations, we will replace the label *A* with 1 for one input valuation and with 2 for the other valuation.

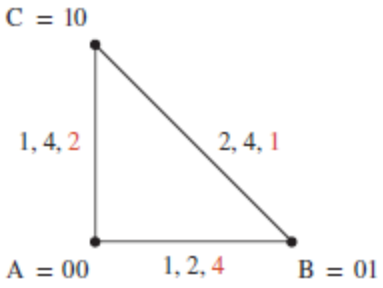
Figure 9.50 shows a relabeled version of the flow table in Figure 9.21*a*. We have arbitrarily chosen to label (A) as 1, the two appearances of (B) as 2 and 3, and the two appearances of (C) as 4 and 5. All entries in each next-state column are labeled using this scheme. The transitions identified by these labels are presented in Figure 9.51*a*. The same information is given in Figure 9.48*a*. Actually, the diagram in Figure 9.48*a* contains more information because arrowheads show the direction of each transition. Note also that the edges in that diagram are labeled with input values r_2r_1, whereas the edges in Figure 9.51*a* are labeled with numerical stable-state labels as explained above.

Present state	Next state				Output
	r_2r_1 = 00	01	10	11	g_2g_1
A	(1)	2	4	–	00
B	1	(2)	4	(3)	01
C	1	2	(4)	(5)	10

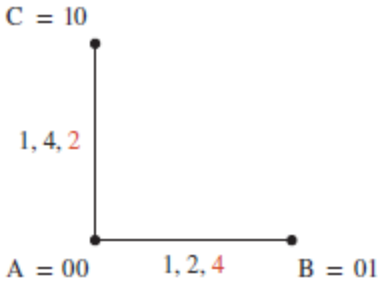
Figure 9.50 Relabeled flow table of Figure 9.21*a*.



(a) Transitions in Figure 9.50



(b) Complete transition diagram



(c) Selected transition diagram

Figure 9.51 Transition diagrams for Figure 9.50.

Figure 9.50 indicates that the stable state 2, which is one instance of the stable state *B*, can be reached either from state *A* or from state *C*. There is a corresponding label 2 on the paths connecting the vertices in the diagram in Figure 9.51*a*. The difficulty from the state-assignment point of view is that the path from *C* to *B* is diagonal. In Example 9.12 this problem was resolved by introducing a new state *D*. By examining the flow table in Figure 9.50 more closely, we can see that the functional behavior of the required arbiter FSM can be achieved if the transition from *C* to *B* takes place via state *A*. Namely, if the circuit is stable in *C*, then the input $r_2\ r_1 = 01$ can cause the change to *A*, from which the circuit immediately proceeds

to state *B*. We can indicate the possibility of using this path by placing the label 2 on the edge that connects *C* and *A* in Figure 9.51*a*.

A similar situation exists for the transition from *B* to *C*, which is labeled 4. An alternative path can be realized by causing the circuit to go from state *B* to state *A* if $r_2\ r_1 = 10$ and then immediately proceed to *C*. This can be indicated by placing the label 4 on the edge that connects *B* and *A* in Figure 9.51*a*.

A possibility of having an alternative path for a transition exists whenever two states have the same uncircled label in the relabeled flow diagram. In Figure 9.50 there is a third such possibility if $r_2\ r_1 = 00$, using the label 1. This possibility is not useful because changing from either *B* or *C* to *A* involves a change in only one state variable using the state assignment in Figure 9.51*a*. Hence there would be no benefit in having a transition between *B* and *C* for this input valuation.

To depict the possibility of having alternative paths, we will indicate in red the corresponding transitions on the diagram. Thus a complete transition diagram will show all direct transitions to stable states in black and possible indirect transitions through unstable states in red. Figure 9.51*b* shows the complete transition diagram for the flow table in Figure 9.21*a*.

The transition diagram in Figure 9.51*b* cannot be embedded on the two-dimensional cube, because some transitions require a diagonal path. The red label 1 on the path between *B* and *C* is of no concern, because it represents only an alternative path that does not have to be used. But the transitions between *B* and *C* labeled 2 and 4 are required. The diagram shows an alternative path, through *A*, having the labels 2 and 4. Therefore, the alternative path can be used, and the diagonal connection in the diagram can be eliminated. This leads to the transition diagram in Figure 9.51*c*, which can be embedded on the two-dimensional cube. The conclusion is that the state assignment $A = 00$, $B = 01$, and $C = 10$ is good, but the flow table must be modified to specify the transitions through alternative paths. The modified table is the same as the flow table designed earlier using an ad hoc approach, shown in Figure 9.23*a*.

As a final comment on this example, note the impact of alternative paths on the outputs produced by the FSM. If $r_2\ r_1 = 01$, then a change from a stable state *C* through unstable *A* to stable *B* generates the outputs $g_2\ g_1 =$

10 → 00 → 01, rather than 10 → 01 as specified in Figure 9.21*a*. For the arbiter FSM this presents no problem, as explained in Example 9.6.

Procedure for Deriving Transition Diagrams

The transition diagram is derived from a flow table as follows:

- Derive the relabeled flow table as explained above. For a given input valuation, all transitions that lead to the same stable state are labeled with the same number. Transitions through unstable states that eventually lead to a stable state are given the same number as the stable-state entry.
- Represent each row of the flow table by a vertex.
- Join two vertices, v_i and v_j, by an edge if they have the same number in any column of the relabeled flow table.
- For each column in which V_i and V_j have the same number, label the edge between V_i and V_j with that number. We will use black labels for direct transitions to circled (stable) states and red labels when the next-state entries for both V_i and V_j in the flow table are uncircled.

Note that the first point says that in the relabeled flow table the transitions through unstable states are given the label of the stable state to which they lead for a given input valuation. For example, to derive a transition diagram starting from the flow table in Figure 9.23*a*, the table would be relabeled to give the table in Figure 9.50. The transition from stable *A* to stable *B*, when $r_2\ r_1 = 01$, has the label 2. The same label is given to the transition from stable *C* to unstable *A* because this transition ultimately leads to stable *B*.

9.5.2 Exploiting Unspecified Next-State Entries

Unspecified entries in a flow table provide some flexibility in finding good state assignments. The following example presents a possible approach. The example also illustrates all steps in the derivation of a transition diagram.

Example 9.13

Consider the flow table in Figure 9.52*a*. This FSM has seven stable-state entries. Labeling these entries in order, from 1 to 7, results in the table in part (*b*) of the figure. In this case states 1 and 2 correspond to state *A*, 3 and 4 to state *B*, 5 and 6 to state *C*, and 7 to state *D*. In the column $w_2\ w_1 = 00$ there is a transition from *C* to *A*, which is labeled 1, and a transition from *D* to *B*, which is labeled 3, because 1 and 3 are the successor stable states in these transitions. Similarly, in column 11 there are transitions from *B* to *C* and from *D* to *A*, which are labeled 6 and 2, respectively. In column 01 there is a transition from *A* to *B*, which is labeled 4. State *C* is stable for this input valuation; it is labeled 5. There is no transition specified that leads to this stable state. The state can be reached only if *C* is stable under $w_2\ w_1 = 11$, which is labeled 6, and then the inputs change to $w_2\ w_1 = 01$. Note that the FSM remains stable in *C* if the inputs change from 11 to 01, or vice versa. Column 10 illustrates how unstable states are treated. From the stable state *A*, a transition to the unstable state *C* is specified. As soon as the FSM reaches state *C*, it proceeds to change to the stable state *D*, which is labeled 7. Thus 7 is used as the label for the entire transition sequence from *A* to *C* to *D*.

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z_2z_1
A	(A)	B	C	(A)	00
B	(B)	(B)	D	C	01
C	A	(C)	D	(C)	10
D	B	–	(D)	A	11

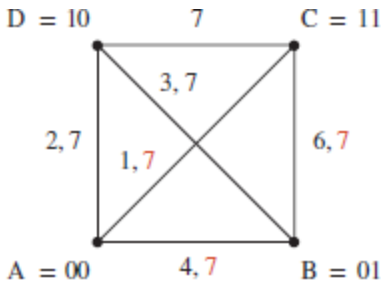
(a) Flow table

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z_2z_1
A	(1)	4	7	(2)	00
B	(3)	(4)	7	6	01
C	1	(5)	7	(6)	10
D	3	–	(7)	2	11

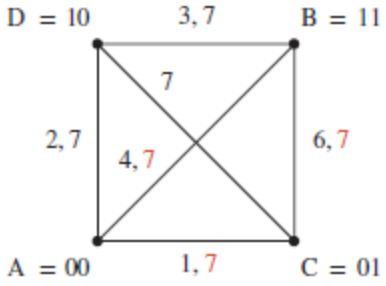
(b) Relabeled flow table

Figure 9.52 Flow tables for Example 9.13.

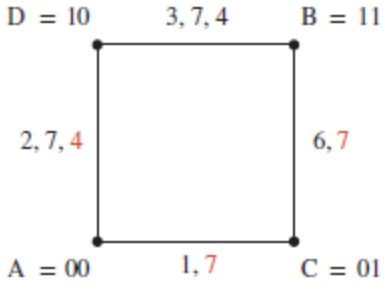
Taking rows *A*, *B*, *C*, and *D* as the four vertices, a first attempt at drawing the transition diagram is given in Figure 9.53*a*. The diagram shows transitions between all pairs of states, which seems to suggest that it is impossible to have a state assignment where all transitions are characterized by a Hamming distance of 1. If the state assignment $A = 00$, $B = 01$, $C = 11$, and $D = 10$ is used, then the diagonal transition between *A* and *C*, or *B* and *D*, requires both state variables to change their values. The diagonal path from *B* to *D* with the label 7 is not needed, because an alternative path from *B* to *D* exists under label 7 that passes either through state *A* or through state *C*. Unfortunately, the diagonal paths labeled 1 and 3 cannot be removed, because there are no alternative paths for these transitions.



(a) First transition diagram



(b) Second transition diagram



(c) Augmented transition diagram

Figure 9.53 Transition diagrams for Figure 9.52.

As the next attempt at finding a suitable state assignment, we will reverse the codes given to *B* and *C*, which yields the transition diagram in Figure 9.53*b*. Now the same argument about the alternative paths labeled 7 indicates that the diagonal from *C* to *D* can be omitted. Also, the label 7 on the diagonal between *A* and *B* can be omitted. However, this diagonal must remain because of the label 4 for which there is no alternative path between *A* and *B*. Looking at the flow table in Figure 9.52*b*, we see an unspecified entry in the column $w_2\, w_1 = 01$. This entry can be exploited by replacing it with the label 4, in which case the transition graph would show the label 4 on the edges connecting *A* and *D*, as well as *B* and *D*. Thus the diagonal

between A and B could be removed, producing the transition diagram in Figure 9.53*c*. This diagram can be embedded on a two-dimensional cube, which means that the state assignment $A = 00$, $B = 11$, $C = 01$, and $D = 10$ can be used.

For the transition diagram in Figure 9.53*c* to be applicable, the flow table for the FSM must be modified as shown in Figure 9.54*a*. The unspecified entry in Figure 9.52*a* now specifies a transition to state B. According to Figure 9.53*c*, the change from state A to B under input valuation $w_2 w_1 = 01$ must pass through state D; hence the corresponding entry in the first row is modified to ensure that this will take place. Also, when $w_2 w_1 = 10$, the FSM must go to state D. If it happens to be in state C, then this change has to occur either via state A or state B. We have chosen the path via state B in Figure 9.54*a*.

Present state	Next state				Output z_2z_1			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	(A)	D	D	(A)	00	00	11	00
B	(B)	(B)	D	C	01	01	11	01
C	A	(C)	B	(C)	−0	10	1−	10
D	B	B	(D)	A	−1	0−	11	00

(a) Modified flow table

	Present state y_2y_1	Next state				Output			
		$w_2w_1 = 00$	01	10	11	00	01	10	11
		Y_2Y_1				z_2z_1			
A	00	(00)	10	10	(00)	00	00	11	00
B	11	(11)	(11)	10	01	01	01	11	01
C	01	00	(01)	11	(01)	−0	10	1−	10
D	10	11	11	(10)	00	−1	0−	11	00

(b) Excitation table

Figure 9.54 Realization of the FSM in Figure 9.52*a*.

The original flow table in Figure 9.52*a* is defined in the form of the Moore model. The modified flow table in Figure 9.54*a* requires the use of the Mealy model because the previously described transitions through unstable states must produce correct outputs. Consider first the change from *A* if $w_2\ w_1 = 01$. While stable in state *A*, the circuit must produce the output $z_2\ z_1 = 00$. Upon reaching the stable state *B*, the output must become 01. The problem is that this transition requires a short visit to state *D*, which in the Moore model would produce $z_2\ z_1 = 11$. Thus a glitch would be generated on the output signal z_2, which would undergo the change $0 \rightarrow 1 \rightarrow 0$. To avoid this undesirable glitch, the output in state *D* must be $z_2 = 0$ for this input valuation, which requires the use of the Mealy model as shown in the Figure 9.54*a*. Observe that while z_2 must be 0 in *D* for $w_2\ w_1 = 01$, z_1 can be either 0 or 1 because it is changing from 0 in state *A* to 1 in state *B*. Therefore, z_1 can be left unspecified so that this case can be treated as a don't-care condition. A similar situation arises when the circuit changes from *C* to *D* via *B* if $w_2\ w_1 = 10$. The output must change from 10 to 11, which means that z_2 must remain at 1 throughout this change, including the short time in state *B* where the Moore model output would be 01.

The modified flow table and the chosen state assignment lead to the excitation table in Figure 9.54*b*. From this table the next-state and output expressions are derived, as in the examples in Section 9.3.

9.5.3 State Assignment Using Additional State Variables

In Figure 9.52*a* there is an unspecified transition that can be exploited to find a suitable state assignment, as shown in Section 9.5.2. In general, such flexibility may not exist. It may be impossible to find a race-free state assignment using $\log_2 n$ state variables for a flow table that has *n* rows. The problem can be solved by adding extra state variables. This can be done in three ways, as illustrated in the examples that follow.

Example 9.14

USING EXTRA UNSTABLE STATES Consider the FSM specified by the flow table in Figure 9.55*a*. The flow table is relabeled in part (*b*) of the figure. A corresponding transition diagram is depicted in Figure 9.56*a*. It indicates that there are transitions between all pairs of vertices (rows). No rearrangement of the existing vertices would allow mapping of the transition diagram onto a two-dimensional cube.

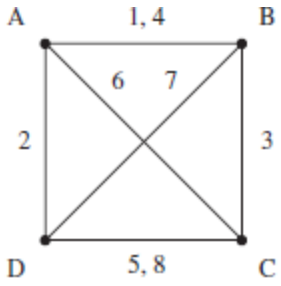
Present state	Next state				Output
	w_2w_1 = 00	01	10	11	z_2z_1
A	(A)	(A)	C	B	00
B	A	(B)	D	(B)	01
C	(C)	B	(C)	D	10
D	C	A	(D)	(D)	11

(a) Flow table

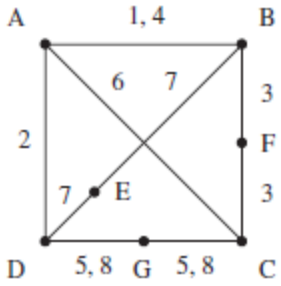
Present state	Next state				Output
	w_2w_1 = 00	01	10	11	z_2z_1
A	(1)	(2)	6	4	00
B	1	(3)	7	(4)	01
C	(5)	3	(6)	8	10
D	5	2	(7)	(8)	11

(b) Relabeled flow table

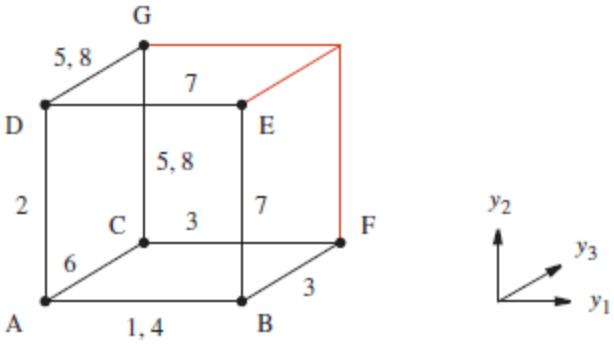
Figure 9.55 FSM for Example 9.14.



(a) Transition diagram



(b) Augmented transition diagram



(c) Embedded transition diagram

Figure 9.56 Transition diagrams for Figure 9.55.

Let us now introduce one more state variable so that we can look for a way to map the transition diagram onto a three-dimensional cube. With three state variables the assignment for state *A* can be a Hamming distance of 1 different from the assignments for *B*, *C*, and *D*. For example, we could have $A = 000$, $B = 001$, $C = 100$, and $D = 010$. But it would then be impossible to have the pairs (*B*, *C*), (*B*, *D*), and (*C*, *D*) within the Hamming distance of 1. The solution here is to insert extra vertices in the transition paths, as shown in Figure 9.56*b*. Vertex *E* separates *B* from *D*, while

vertices F and G break the paths (B, C) and (C, D). The labels associated with the transitions are attached to both segments of a broken path. The resulting transition diagram can be embedded onto a three-dimensional cube as indicated in Figure 9.56*c*, where the black portion of the cube comprises the desired paths. Now the transition from B to D takes place via vertex E if $w_2\ w_1 = 10$ (label 7). The transition from C to B occurs via F if $w_2\ w_1 = 01$ (label 3). The transition from C to D goes through G if $w_2\ w_1 = 11$ (label 8), and the transition from D to C goes via G if $w_2\ w_1 = 00$ (label 5). Therefore, the flow table has to be modified as shown in Figure 9.57*a*. The three extra states are unstable because the circuit will not remain in these states for any valuation of the inputs. The circuit will merely pass through these states in the process of changing from one stable state to another. Observe that each of the states E, F, and G is needed to facilitate the transitions caused by only one or two valuations of inputs. Thus it is not necessary to specify the actions that might be caused by other input valuations, because such situations will never occur in a properly functioning circuit.

The outputs in Figure 9.57*a* can be specified using the Mealy model. It is essential that a proper output is generated when passing through unstable states, to avoid undesirable glitches in the output signals.

If we assign the state variables as shown on the right of Figure 9.56*c*, the modified flow table leads to the excitation table in Figure 9.57*b*. From this table, deriving the next-state and output expressions is a straightforward task.

Present state	Next state w_2w_1 = 00	01	10	11	Output z_2z_1
A	(A)	(A)	C	B	00
B	A	(B)	E	(B)	01
C	(C)	F	(C)	G	10
D	G	A	(D)	(D)	11
E	–	–	D	–	– 1
F	–	B	–	–	01
G	C	–	–	D	1–

(a) Modified flow table

	Present state $y_3y_2y_1$	Next state w_2w_1 = 00	01	10	11	Output z_2z_1
		$Y_3Y_2Y_1$				
A	000	(000)	(000)	100	001	00
B	001	000	(001)	011	(001)	01
C	100	(100)	101	(100)	110	10
D	010	110	000	(010)	(010)	11
E	011	–	–	010	–	– 1
F	101	–	001	–	–	01
G	110	100	–	–	010	1–

(b) Excitation table

Figure 9.57 Modified tables for Example 9.14.

Example 9.15

USING PAIRS OF EQUIVALENT STATES Another approach is to increase the flexibility in state assignment by introducing an equivalent new state for each existing state. Thus state A can be replaced with two states $A1$ and $A2$ such that the final circuit produces the same outputs for $A1$ and $A2$ as it would for A. Similarly, other states can be replaced by equivalent pairs of states. Figure 9.58 shows how a three-dimensional cube can be used to find

a good state assignment for a four-row flow table. The four equivalent pairs are arranged so that the minimum Hamming distance of 1 exists between all pairs. For example, the pair ($B1$, $B2$) has the Hamming distance of 1 with respect to $A1$ (or $A2$), $C2$, and $D2$.

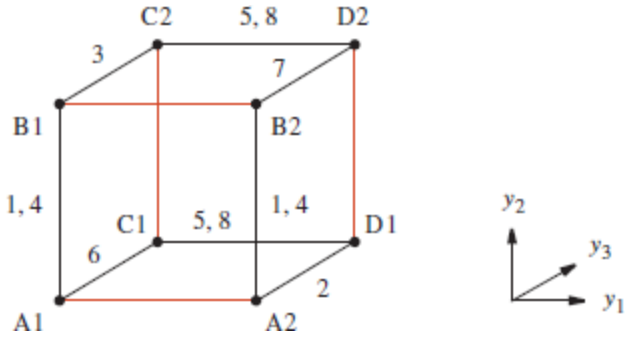


Figure 9.58 Embedded transition diagram if two nodes per row are used.

The transition diagram in Figure 9.56*a* can be embedded onto the three-dimensional cube as shown in Figure 9.58. Since there is a choice of two vertices on the cube for each vertex in the transition diagram in Figure 9.56*a*, the embedded transition diagram does not involve any diagonal paths. Using this assignment of states, the flow table in Figure 9.55*a* has to be modified as presented in Figure 9.59*a*. The entries in the table are made to allow each transition in the original flow table to be realized using a transition between the corresponding pairs of equivalent states. Both states in an equivalent pair are stable for the input valuations for which the original state is stable. Thus $A1$ and $A2$ are stable if $w_2\ w_1 = 00$ or 01, $B1$ and $B2$ are stable if $w_2\ w_1 = 01$ or 11, and so on. At any given time the FSM may be in either of the two equivalent states that represent an original state. Then a change to another state must be possible from either of these states. For example, Figure 9.55*a* specifies that the FSM must change from the stable state A to state B if the input is $w_2\ w_1 = 11$. The equivalent transition in the modified flow table is the change from state $A1$ to $B1$ or from state $A2$ to $B2$. If the FSM is stable in A and the input changes from 00 to 10, then a change to C is required. The equivalent transition in the modified flow table is from state $A1$ to $C1$; if the FSM happens to be in state $A2$, it will first have to change to $A1$. The remaining entries in Figure 9.59*a* are derived using the same reasoning.

Present state	Next state w_2w_1 = 00	01	10	11	Output z_2z_1
A1	(A1)	(A1)	C1	B1	00
A2	(A2)	(A2)	A1	B2	00
B1	A1	(B1)	B2	(B1)	01
B2	A2	(B2)	D2	(B2)	01
C1	(C1)	C2	(C1)	D1	10
C2	(C2)	B1	(C2)	D2	11
D1	C1	A2	(D1)	(D1)	11
D2	C2	D1	(D2)	(D2)	11

(a) Modified flow table

	Present state $y_3y_2y_1$	Next state w_2w_1 = 00	01	10	11	Output z_2z_1
		$Y_3Y_2Y_1$				
A1	000	(000)	(000)	100	010	00
A2	001	(001)	(001)	000	011	00
B1	010	000	(010)	011	(010)	01
B2	011	001	(011)	111	(011)	01
C1	100	(100)	110	(100)	101	10
C2	110	(110)	010	(110)	111	10
D1	101	100	001	(101)	(101)	11
D2	111	110	101	(111)	(111)	11

(b) Excitation table

Figure 9.59 Modified flow and excitation tables for Example 9.15.

The outputs are specified using the Moore model, because the only unstable states are those involved in changing from one member of the equivalent pair to another, and both members generate the same outputs. For instance, in the previously described transition from *A* to *C*, if the

starting point is $A2$, it is necessary to go first to $A1$ and then to $C1$. Even though $A1$ is unstable for $w_2\ w_1 = 10$, there is no problem because its output is the same as that of $A2$. Therefore, if the original flow table is defined using the Moore model, then the modified flow table can also be done using the Moore model.

Using the assignment of the state variables in Figure 9.58 gives the excitation table in Figure 9.59*b*.

9.5.4 ONE-HOT STATE ASSIGNMENT

The previously described schemes based on embedding the flow table in a cube may lead to an optimal state assignment, but they require a trial-and-error approach that becomes awkward for large machines. A straightforward, but more expensive, alternative is to use one-hot codes. If each row in the flow table of an FSM is assigned a one-hot code, then race-free state transitions can be achieved by passing through unstable states that are at a Hamming distance of 1 from the two stable states involved in the transition. For example, suppose that state A is assigned the code 0001 and state B the code 0010. Then a race-free transition from A to B can pass through an unstable state 0011. Similarly, if C is assigned the code 0100, then a transition from A to C can be done via the unstable state 0101.

Using this approach, the flow table in Figure 9.55*a* can be modified as illustrated in Figure 9.60. The four states, A, B, C, and D, are assigned one-hot codes. As seen in the figure, it is necessary to introduce six unstable states, E through J, to handle the necessary transitions. These unstable states have to be specified only for the specific transitions, whereas for other input valuations they may be treated as don't cares.

State assignment	Present state	Next state w_2w_1 = 00	01	10	11	Output z_2z_1
0001	A	(A)	(A)	E	F	00
0010	B	F	(B)	G	(B)	01
0100	C	(C)	H	(C)	I	10
1000	D	I	J	(D)	(D)	11
0101	E	–	–	C	–	– 0
0011	F	A	–	–	B	0 –
1010	G	–	–	D	–	– 1
0110	H	–	B	–	–	01
1100	I	C	–	–	D	1 –
1001	J	–	A	–	–	00

Figure 9.60 State assignment with one-hot encoding.

The outputs can be specified using the Moore model. In some cases it does not matter when a particular output signal changes its value. For instance, state *E* is used to facilitate the transition from state *A* to *C*. Since $z_2\ z_1 = 00$ in *A* and 10 in *C*, it is not important if z_2 changes when passing through state *E*.

While straightforward to implement, the one-hot encoding is expensive because it requires *n* state variables to implement an *n*-row flow table. Simplicity of design and the cost of implementation often provide a challenging trade-off in designing logic circuits!

9.6 HAZARDS

In asynchronous sequential circuits it is important that undesirable glitches on signals should not occur. The designer must be aware of the possible sources of glitches and ensure that the transitions in a circuit will be glitch free. The glitches caused by the structure of a given circuit and propagation delays in the circuit are referred to as *hazards*. Two types of hazards are illustrated in Figure 9.61.

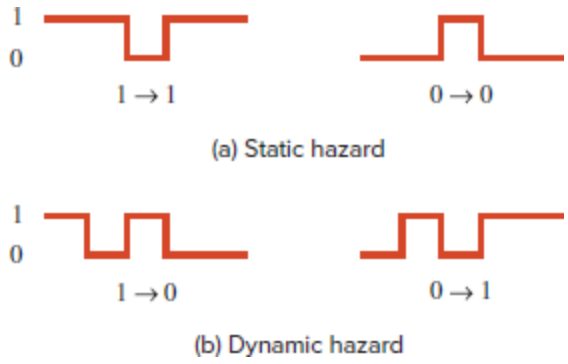


Figure 9.61 Definition of hazards.

A *static hazard* exists if a signal is supposed to remain at a particular logic value when an input variable changes its value, but instead the signal undergoes a momentary change in its required value. As shown in Figure 9.61*a*, one type of static hazard is when the signal at level 1 is supposed to remain at 1 but dips to 0 for a short time. Another type is when the signal is supposed to remain at level 0 but rises momentarily to 1, thus producing a glitch.

A different type of hazard may occur when a signal is supposed to change from 1 to 0 or from 0 to 1. If such a change involves a short oscillation before the signal settles into its new level, as illustrated in Figure 9.61*b*, then a *dynamic hazard* is said to exist.

9.6.1 STATIC HAZARDS

Figure 9.62*a* shows a circuit with a static hazard. Suppose that the circuit is in the state where $x_1 = x_2 = x_3 = 1$, in which case $f = 1$. Now let x_1 change from 1 to 0. Then the circuit is supposed to maintain $f = 1$. But consider what happens when the propagation delays through the gates are taken into account. The change in x_1 will probably be observed at point p before it will be seen at point q because the path from x_1 to q has an extra gate (NOT) in it. Thus the signal at p will become 0 before the signal at q becomes equal to 1. For a short time both p and q will be 0, causing f to drop to 0 before it recovers back to 1. This gives rise to the signal depicted on the left side of Figure 9.61*a*.

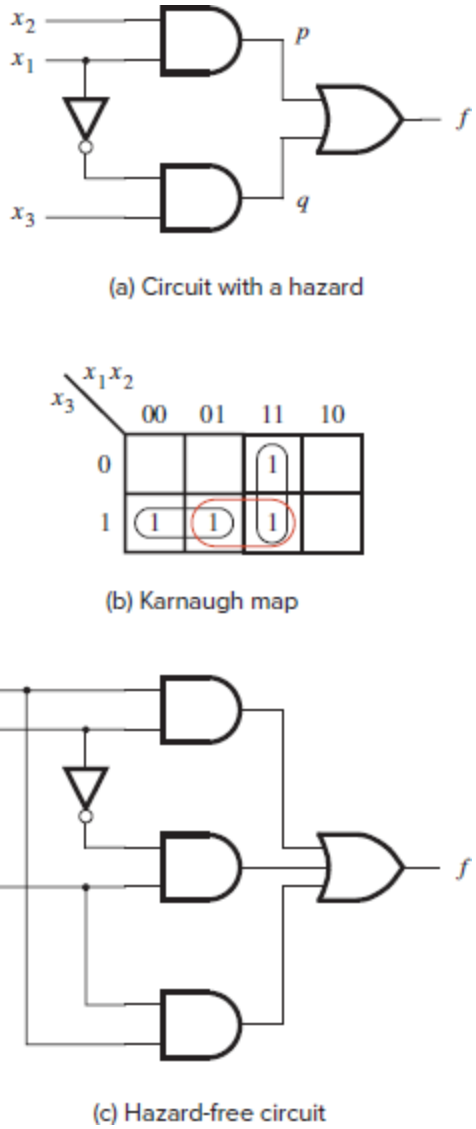


Figure 9.62 An example of a static hazard.

The glitch on f can be prevented as follows. The circuit implements the function

$$f = x_1x_2 + \bar{x}_1x_3$$

The corresponding Karnaugh map is given in Figure 9.62*b*. The two product terms realize the prime implicants encircled in black. The hazard explained above occurs when there is a transition from the prime implicant x_1x_2 to the prime implicant $\bar{x}_1x_3$. The hazard can be eliminated by including the third prime implicant, encircled in red. (This is the consensus term,

defined in Property 17*a* in Section 2.5.) Then the function would be implemented as

$$f = x_1x_2 + \bar{x}_1x_3 + x_2x_3$$

Now the change in x_1 from 1 to 0 would have no effect on the output f because the product term x_2x_3 would be equal to 1 if $x_2 = x_3 = 1$, regardless of the value of x_1. The resulting hazard-free circuit is depicted in Figure 9.62*c*.

A potential hazard exists wherever two adjacent 1s in a Karnaugh map are not covered by a single product term. Therefore, a technique for removing hazards is to find a cover in which some product term includes each pair of adjacent 1s. Then, since a change in an input variable causes a transition between two adjacent 1s, no glitch can occur because both 1s are included in a product term.

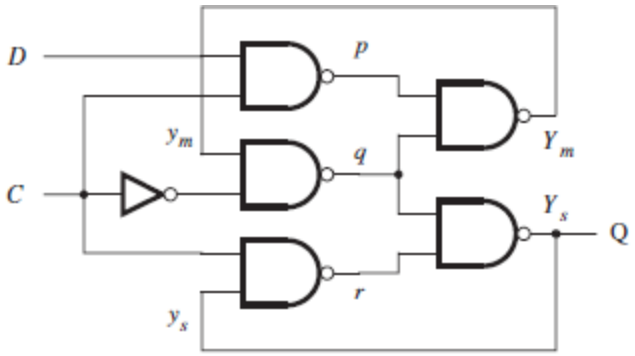
In asynchronous sequential circuits a hazard can cause the circuit to change to an incorrect stable state. Example 9.16 illustrates this situation.

Example 9.16

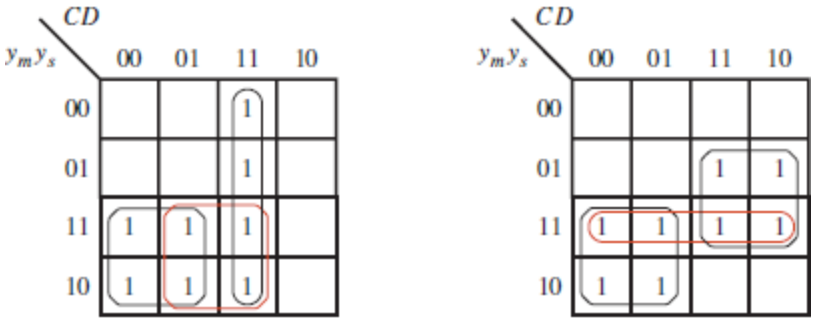
In Example 9.2 we analyzed the circuit that realizes a M-S D flip-flop. From the excitation table in Figure 9.6*a*, one could attempt to synthesize a minimum-cost circuit that realizes the required functions, Y_m and Y_s. This would give

$$\begin{aligned} Y_m &= CD + \bar{C}y_m \\ &= (C \uparrow D) \uparrow (\bar{C} \uparrow y_m) \\ Y_s &= \bar{C}y_m + Cy_s \\ &= (\bar{C} \uparrow y_m) \uparrow (C \uparrow y_s) \end{aligned}$$

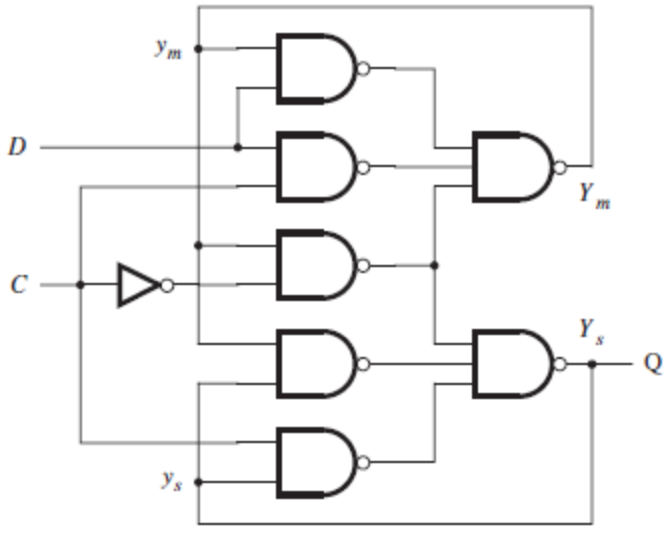
The corresponding circuit is presented in Figure 9.63*a*. At first glance this circuit may seem more attractive than the flip-flops discussed in Chapter 5 because it is less expensive. The problem is that the circuit contains a static hazard.



(a) Minimum-cost circuit



(b) Karnaugh maps for Y_m and Y_s in Figure 9.6*a*



(c) Hazard-free circuit

Figure 9.63 Two-level implementation of M-S flip-flop.

Figure 9.63*b* shows the Karnaugh maps for the functions Y_m and Y_s. The minimum-cost implementation is based on the prime implicants encircled in black. To see how this circuit is affected by static hazards, assume that

presently $Y_s = 1$ and $C = D = 1$. The circuit generates $Y_m = 1$. Now let C change from 1 to 0. For the flip-flop to behave properly, Y_s must remain equal to 1. In Figure 9.63*a*, when C changes to 0, both p and r become 1. Due to the delay through the NOT gate, q may still be 1, causing the circuit to generate $Y_m = Y_s = 0$. The feedback from Y_m will maintain $q = 1$. Hence the circuit remains in an incorrect stable state with $Y_s = 0$.

To avoid the hazards, it is necessary to also include the terms encircled in red, which gives rise to the expressions

$$Y_m = CD + \overline{C}y_m + Dy_m$$
$$Y_s = \overline{C}y_m + Cy_s + y_m y_s$$

The resulting circuit, implemented with NAND gates, is shown in Figure 9.63*c*.

Note that we can obtain another NAND-gate implementation by rewriting the expressions for Y_m and Y_s as

$$\begin{aligned} Y_m &= CD + (\overline{C} + D)y_m \\ &= (C \uparrow D) \uparrow ((\overline{C} + D) \uparrow y_m) \\ &= (C \uparrow D) \uparrow ((C \uparrow \overline{D}) \uparrow y_m) \\ Y_s &= \overline{C}y_m + (C + y_m)y_s \\ &= (\overline{C} \uparrow y_m) \uparrow ((\overline{C} \uparrow \overline{y}_m) \uparrow y_s) \end{aligned}$$

These expressions correspond exactly to the circuit in Figure 5.12 (without the clear and preset inputs).

Example 9.17

From the previous examples, it seems that static hazards can be avoided by including all prime implicants in a sum-of-products circuit that realizes a given function. This is indeed true. But it is not always necessary to include all prime implicants. It is only necessary to include product terms that cover the adjacent pairs of 1s. There is no need to cover the don't-care vertices.

Consider the function in Figure 9.64. A hazard-free circuit that implements this function should include the encircled terms, which gives

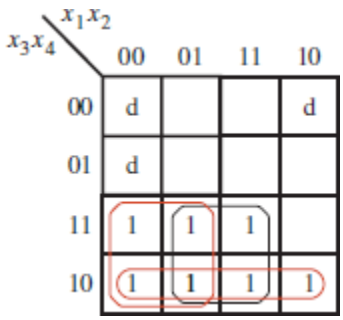


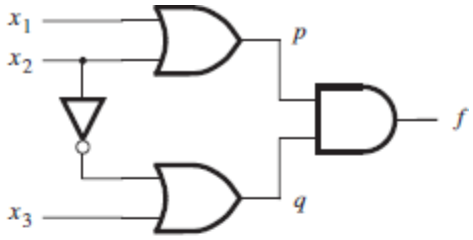
Figure 9.64 Function for Example 9.17.

$$f = \bar{x}_1 x_3 + x_2 x_3 + x_3 \bar{x}_4$$

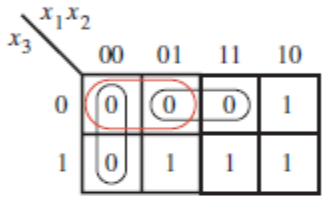
The prime implicant $\bar{x}_1 \bar{x}_2$ is not needed to prevent hazards, because it would account only for the two 1s in the left-most column. These 1s are already covered by $\bar{x}_1 x_3$.

Example 9.18

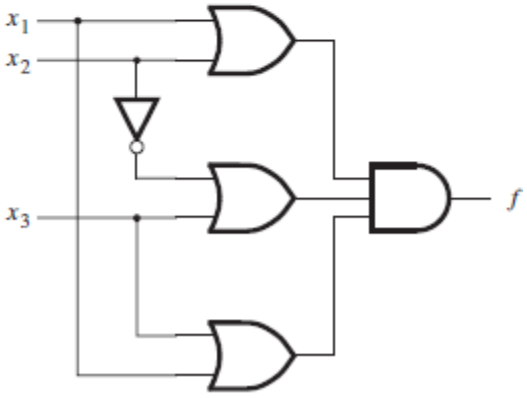
static hazards can also occur in other types of circuits. Figure 9.65*a* depicts a product-of-sums circuit that contains a hazard. If $x_1 = x_3 = 0$ and x_2 changes from 0 to 1, then f should remain at 0. However, if the signal at p changes earlier than the signal at q, then p and q will both be equal to 1 for a short time, causing a glitch $0 \rightarrow 1 \rightarrow 0$ on f.



(a) Circuit with a hazard



(b) Karnaugh map



(c) Hazard-free circuit

Figure 9.65 Static hazard in a POS circuit.

In a POS circuit, it is the transitions between adjacent 0s that may lead to hazards. Thus to design a hazard-free circuit, it is necessary to include sum terms that cover all pairs of adjacent 0s. In this example the term in red in the Karnaugh map must be included, giving

$$f = (x_1 + x_2)(\overline{x}_2 + x_3)(x_1 + x_3)$$

The circuit is shown in Figure 9.65*c*.

9.6.2 DYNAMIC HAZARDS

A dynamic hazard causes glitches on $0 \rightarrow 1$ or $1 \rightarrow 0$ transitions of an output signal. An example is given in Figure 9.66. Assuming that all NAND gates have equal delays, a timing diagram can be constructed as shown. The time elapsed between two vertical lines corresponds to a gate delay. The output f exhibits a glitch that should be avoided.

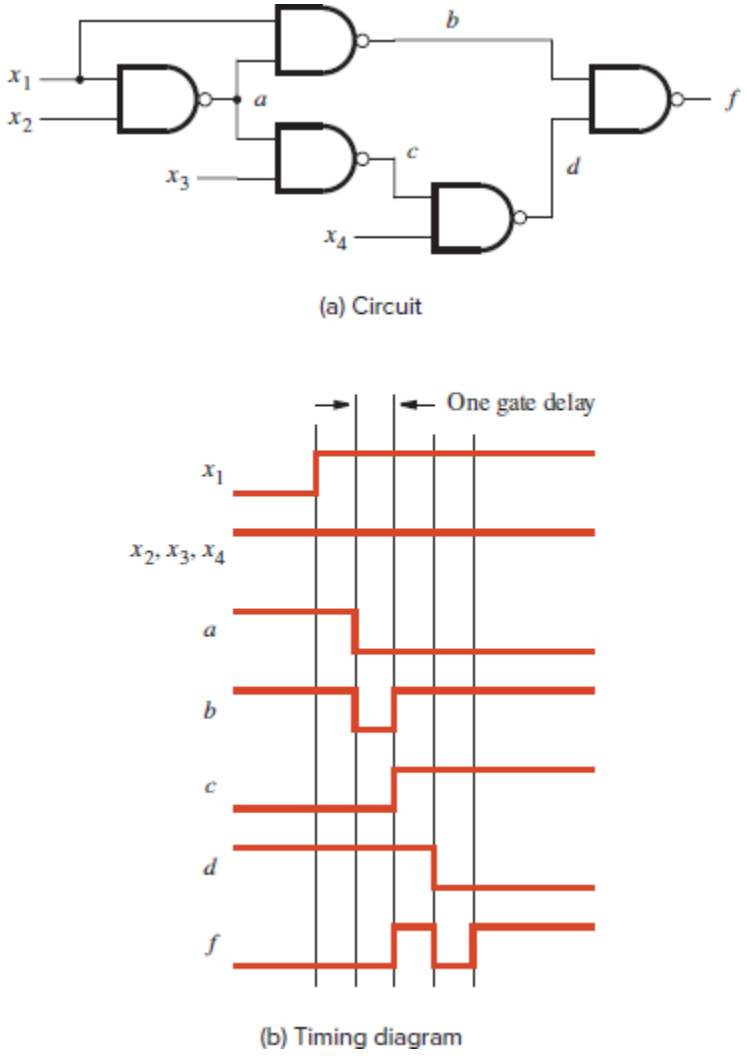


Figure 9.66 Circuit with a dynamic hazard.

It is interesting to consider the function implemented by this circuit, which is

$$f = x_1\bar{x}_2 + \bar{x}_3x_4 + x_1x_4$$

This is the minimum-cost sum-of-products expression for the function. If implemented in this form, the circuit would not have either a static or a dynamic hazard.

A dynamic hazard is caused by the structure of the circuit, where there exist multiple paths for a given signal change to propagate along. If the output signal changes its value three times, $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ in the example, then there must be at least three paths along which a change from a primary input can propagate. A circuit that has a dynamic hazard must also have a static hazard in some part of it. As seen in Figure 9.66*b*, there is a static hazard involving the signal on wire *b*.

Dynamic hazards are encountered in multilevel circuits obtained using factoring or decomposition techniques, which were discussed in Chapter 8. Such hazards are neither easy to detect nor easy to deal with. The designer can avoid dynamic hazards simply by using two-level circuits and ensuring that there are no static hazards.

9.6.3 Significance of Hazards

A glitch in an asynchronous sequential circuit can cause the circuit to enter an incorrect state and possibly become stable in that state. Therefore, the circuitry that generates the next-state variables must be hazard free. It is sufficient to eliminate hazards due to changes in the value of a single variable because the basic premise in an asynchronous sequential circuit is that the values of both the primary inputs and the state variables must change one at a time.

In combinational circuits, discussed in Chapters 3 and 4, we did not worry about hazards, because the output of a circuit depends solely on the values of the inputs. In synchronous sequential circuits the input signals must be stable within the setup and hold times of flip-flops. It does not matter whether glitches occur outside the setup and hold times with respect to the clock signal.

9.7 A Complete Design Example

In the previous sections we examined the various design aspects of asynchronous sequential circuits. In this section we give a complete design

example, which covers all necessary steps.

9.7.1 THE VENDING-MACHINE CONTROLLER

The control mechanism of a vending machine is a good vehicle for illustrating a possible application of a digital circuit. We used it in the synchronous environment in Chapter 6. A small example of a vending machine served as an object of analysis in Section 9.2. Now we will consider a vending-machine controller similar to the one in Example 6.6 to see how it can be implemented using an asynchronous sequential circuit. The specification for the controller is:

- It accepts nickels and dimes.
- A total of 15 cents is needed to release the candy from the machine.
- No change is given if 20 cents is deposited.

Coins are deposited one at a time. The coin-sensing mechanism generates signals $N = 1$ and $D = 1$ when it sees a nickel or a dime, respectively. It is impossible to have $N = D = 1$ at the same time. Following the insertion of a coin for which the sum equals or exceeds 15 cents, the machine releases the candy and resets to the initial state.

Figure 9.67 shows a state diagram for the required FSM. It is derived using a straightforward approach in which all possible sequences of depositing nickels and dimes are enumerated in a treelike structure. To keep the diagram uncluttered, the labels *D* and *N* denote the input conditions *D* N = 10 and *D* N = 01, respectively. The condition *D* N = 00 is labeled simply as 0. The candy is released in states *F*, *H*, and *K*, which are reached after 15 cents has been deposited, and in states *I* and *L*, upon a deposit of 20 cents.

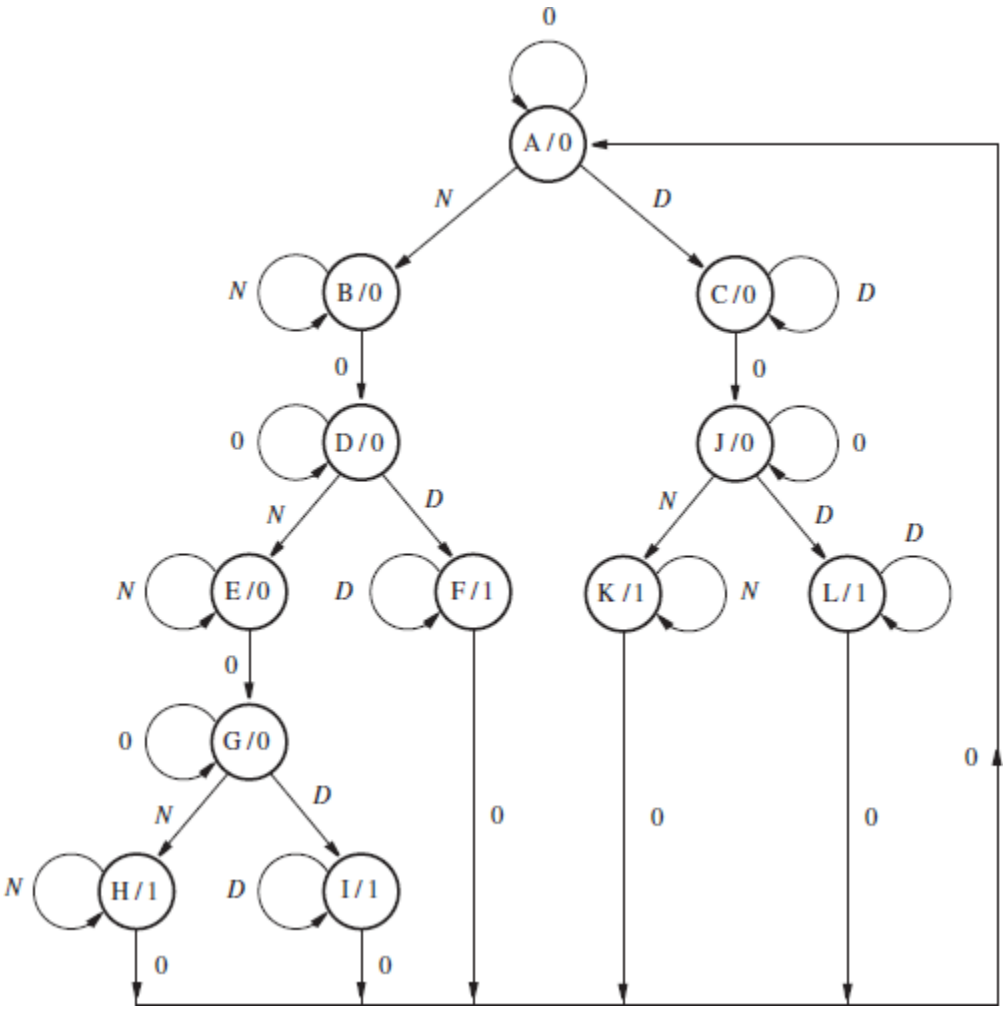


Figure 9.67 Initial state diagram for the vending-machine controller.

The corresponding flow table is given in Figure 9.68. It can be reduced using the partitioning procedure as follows

$$P_1 = (ADGJ)(BE)(C)(FIL)(HK)$$
$$P_2 = (A)(D)(GJ)(B)(E)(C)(FIL)(HK)$$
$$P_3 = P_2$$

Using G to represent the equivalent states G and J, F to represent F, I, and L, and H to represent H and K yields a partially reduced flow table in Figure 9.69. The merger diagram for this table is presented in Figure 9.70. It indicates that states C and E can be merged, as well as F and H. Thus the reduced flow table is obtained as shown in Figure 9.71*a*. The same information is depicted in the form of a state diagram in Figure 9.72.

Present state	Next state				Output
	DN = 00	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	J	–	(C)	–	0
D	(D)	E	F	–	0
E	G	(E)	–	–	0
F	A	–	(F)	–	1
G	(G)	H	I	–	0
H	A	(H)	–	–	1
I	A	–	(I)	–	1
J	(J)	K	L	–	0
K	A	(K)	–	–	1
L	A	–	(L)	–	1

Figure 9.68 Initial flow table for the vending-machine controller.

Present state	Next state				Output
	DN = 00	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	G	–	(C)	–	0
D	(D)	E	F	–	0
E	G	(E)	–	–	0
F	A	–	(F)	–	1
G	(G)	H	F	–	0
H	A	(H)	–	–	1

Figure 9.69 First step in state minimization.

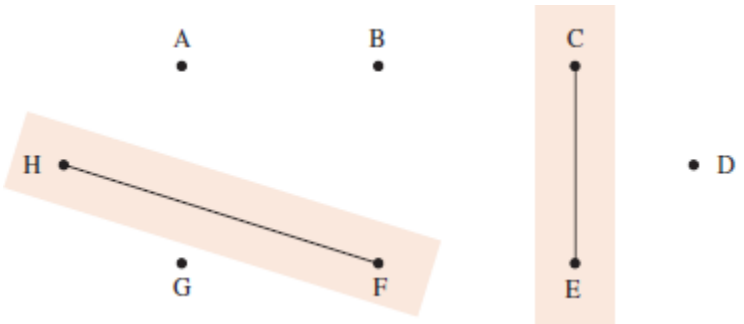


Figure 9.70 Merger diagram for Figure 9.69.

Present state	Next state				Output
	DN = 00	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	G	(C)	(C)	–	0
D	(D)	C	F	–	0
F	A	(F)	(F)	–	1
G	(G)	F	F	–	0

(a) Minimized flow table

Present state	Next state				Output
	DN = 00	01	10	11	z
A	(1)	2	4	–	0
B	5	(2)	–	–	0
C	8	(3)	(4)	–	0
D	(5)	3	7	–	0
F	1	(6)	(7)	–	1
G	(8)	6	7	–	0

(b) Relabeled flow table

Figure 9.71 Reduced flow tables.

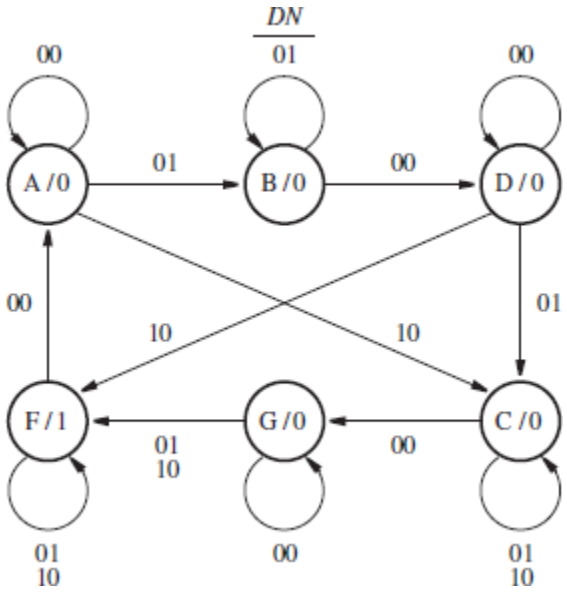


Figure 9.72 State diagram for the vending-machine controller.

Next a suitable state assignment must be found. The flow table is relabeled in Figure 9.71*b* to associate a unique number with each stable state. Then the transition diagram in Figure 9.73*a* is obtained. Since we wish to try to embed the diagram onto a three-dimensional cube, eight vertices are shown in the figure. The diagram shows two diagonal transitions. The transition between *D* and *G* (label 7) does not matter, because it is only an alternative path. The transition from *A* to *C* (label 4) is required, and it can be realized via unused states as indicated in red in Figure 9.73*b*. Therefore, the transition diagram can be embedded onto a three-dimensional cube as shown. Using the state assignment from this figure, the excitation table in Figure 9.74 is derived.

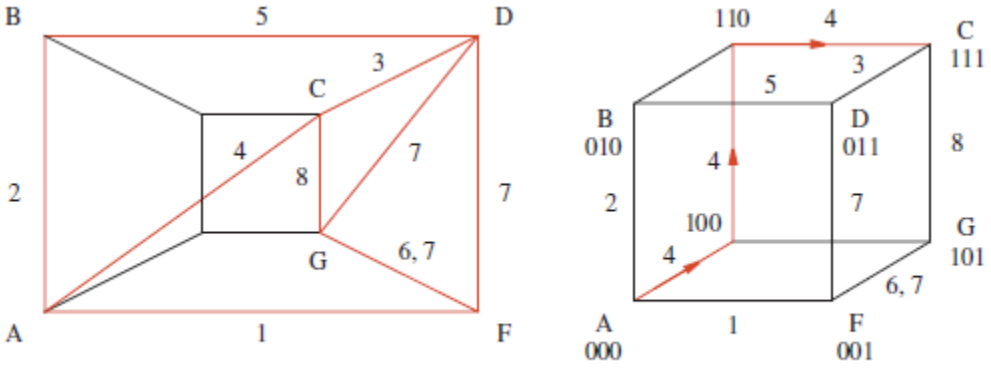


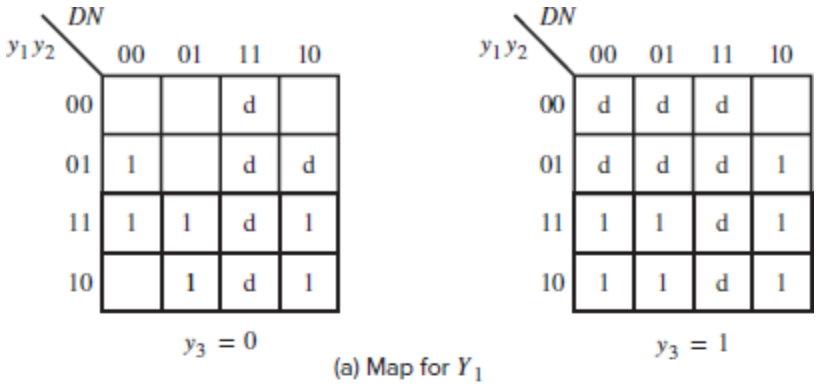
Figure 9.73 Determination of the state assignment.

	Present state $y_3y_2y_1$	Next state $Y_3Y_2Y_1$				Output z
		$DN = 00$	01	10	11	
A	000	(000)	010	100	–	0
B	010	011	(010)	–	–	0
C	111	101	(111)	(111)	–	0
D	011	(011)	111	001	–	0
F	001	000	(001)	(001)	–	1
G	101	(101)	001	001	–	0
	100	–	–	110	–	0
	110	–	–	111	–	0

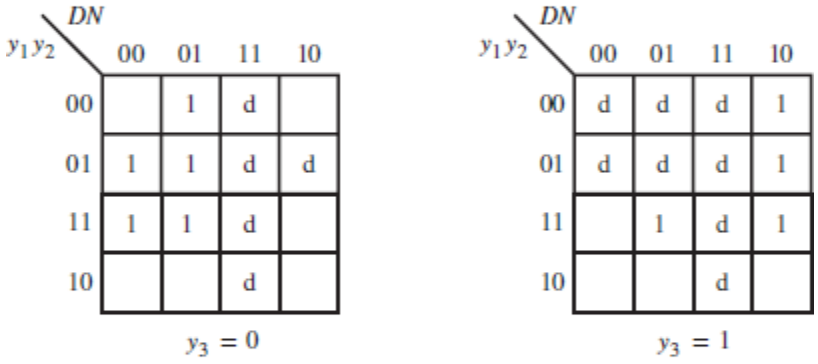
Figure 9.74 Excitation table based on the state assignment in Figure 9.73*b*.

The Karnaugh maps for the next-state functions are given in Figure 9.75. From these maps the following hazard-free expressions are obtained

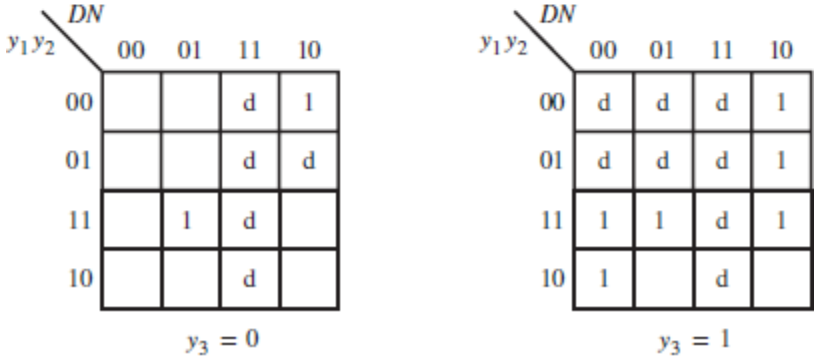
$$Y_1 = \overline{N}y_2 + Ny_1 + Dy_1 + y_1y_3 + y_1y_2$$
$$Y_2 = N\overline{y}_1 + Ny_2 + \overline{y}_1y_3 + \overline{D}y_2\overline{y}_3 + Dy_2y_3$$
$$Y_3 = D\overline{y}_1 + y_2y_3 + Ny_1y_2 + \overline{D}y_3\overline{N}$$



(a) Map for Y_1



(b) Map for Y_2



(c) Map for Y_3

Figure 9.75 Karnaugh maps for the function in Figure 9.74.

All product terms in these expressions are needed for a minimum-cost POS implementation except for y_1y_2, which is included to prevent hazards in the expression for Y_1. The output expression is

$$z = y_1\bar{y}_2\bar{y}_3$$

9.8 Concluding Remarks

Asynchronous sequential circuits are more difficult to design than the synchronous sequential circuits. The difficulties with race conditions present a problem that must be handled carefully. At the present time there is little CAD support for designing asynchronous circuits. For these reasons, most designers resort to synchronous sequential circuits in practical applications.

An important advantage of asynchronous circuits is their speed of operation. Since there is no clock involved, the speed of operation depends only on the propagation delays in the circuit. In an asynchronous system that comprises several circuits, some circuits may operate faster than others, thus potentially improving the overall performance of the system. In contrast, in synchronous systems the clock period has to be long enough to accommodate the slowest circuit, and it has a large effect on the performance.

Asynchronous circuit techniques are also useful in designing systems that consist of two or more synchronous circuits that operate under the control of different clocks. The signals exchanged between such circuits often appear to be asynchronous in nature.

From the reader's point of view, it is useful to view asynchronous circuits as an excellent vehicle for gaining a deeper understanding of the operation of digital circuits in general. These circuits illustrate the consequences of propagation delays and race conditions that may be inherent in the structure of a circuit. They also illustrate the concept of stability, demonstrated through the existence of stable and unstable states.

9.9 Examples of Solved Problems

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

Example 9.19

Problem: Derive a flow table that describes the behavior of the circuit in Figure 9.76.

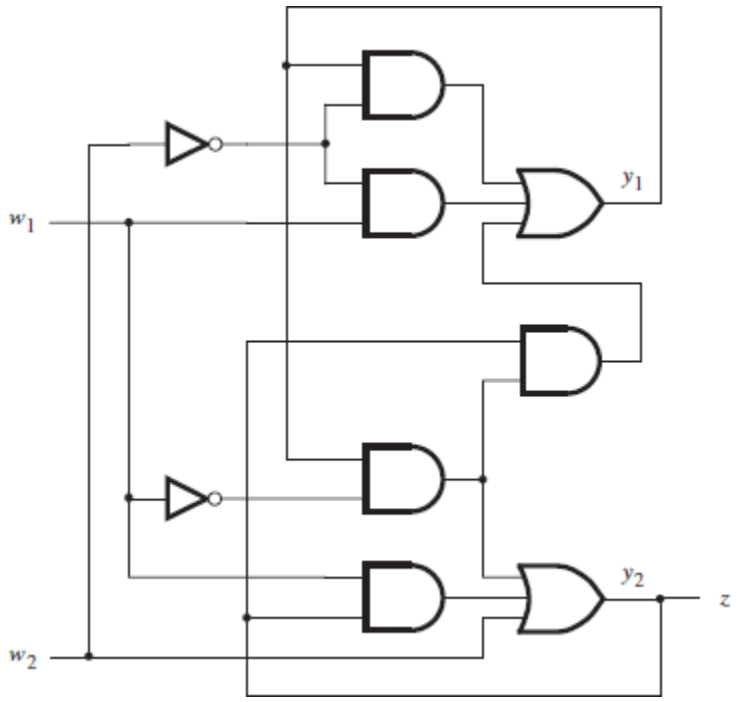


Figure 9.76 Circuit for Example 9.19.

Solution: Modeling the propagation delay in the gates of the circuit as shown in Figure 9.8, the circuit in Figure 9.76 can be described by the following next-state and output expressions

$$Y_1 = w_1\overline{w}_2 + \overline{w}_2 y_1 + \overline{w}_1 y_1 y_2$$
$$Y_2 = w_2 + \overline{w}_1 y_1 + w_1 y_2$$
$$z = y_2$$

These expressions lead to the excitation table in Figure 9.77*a*. Assuming the state assignment $A = 00$, $B = 01$, $C = 10$, and $D = 11$, yields the flow table in Figure 9.77*b*.

Present state $y_2 y_1$	Next state w_2w_1 = 00	01	10	11	Output z
	Y_2Y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	
00	(00)	01	10	10	0
01	11	(01)	10	10	0
10	00	11	(10)	(10)	1
11	(11)	(11)	(11)	10	1

(a) Excitation table

Present state	Next state w_2w_1 = 00	01	10	11	Output z
A	(A)	B	C	C	0
B	D	(B)	C	C	0
C	A	D	(C)	(C)	1
D	(D)	(D)	(D)	C	1

(b) Flow table implemented by the circuit

Present state	Next state w_2w_1 = 00	01	10	11	Output z
A	(A)	B	C	C	0
B	D	(B)	–	C	0
C	A	D	(C)	(C)	1
D	(D)	(D)	(D)	C	1

(c) Final flow table

Figure 9.77 Excitation and flow tables for the circuit in Figure 9.76.

Since in a given stable state, inputs to the circuit can only change one at a time, some entries in the flow table may be designated as unspecified. Such is the case when the circuit is stable in state *B* and input values are $w_2\ w_1$ = 01. Now, both inputs cannot change simultaneously, which means that the corresponding entry in the flow table should be designated as unspecified. However, a different situation arises when the circuit is stable in state *A* and input values are $w_2\ w_1$ = 00. In this case, we cannot indicate the transition in

column w_2 w_1 = 11 as unspecified. The reason is that if the circuit is in stable state *B*, it has to be able to change to state *C* when w_2 changes from 0 to 1. States *B* and *C* are implemented as y_2 y_1 = 01 and y_2 y_1 = 10, respectively. Since both state variables must change their values, the route from 01 to 10 will take place either via 11 or 00, depending on the delays in different paths in the circuit. If y_2 changes first, the circuit will pass through unstable state *D* and then settle in the stable state *C*. But, if w_1 changes first, the circuit has to pass through unstable state *A* before reaching state *C*. Hence, the transition to state *C* in the first row must be specified. This is an example of a safe race, where the circuit reaches the correct destination state regardless of propagation delays in different paths of the circuit. The final flow table is presented in Figure 9.77*c*.

Example 9.20

Problem: Are there any hazards in the circuit in Figure 9.76?

Solution: Figure 9.78 gives Karnaugh maps for the next-state expressions derived in Example 9.19. As seen from the maps, all prime implicants are included in the expression for Y_1. But, the expression for Y_2 includes only three of the four available prime implicants. There is a static hazard when w_2 y_2 y_1 = 011 and w_1 changes from 0 to 1 (or 1 to 0). This hazard can be removed by adding the fourth prime implicant, y_1y_2, to the expression for Y_2.

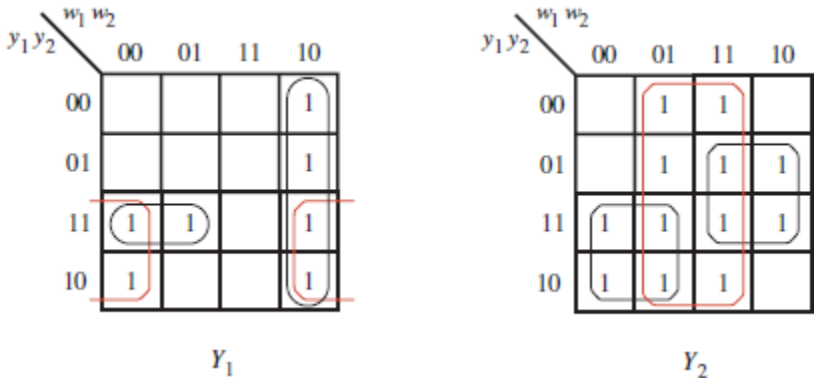


Figure 9.78 Karnaugh maps for the circuit in Figure 9.76.

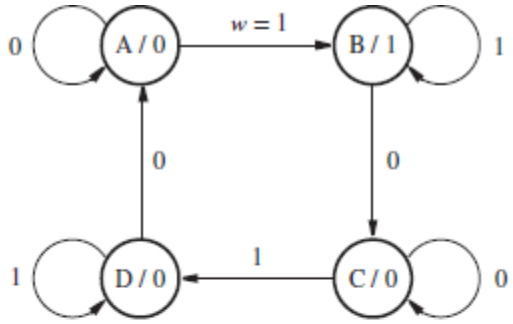
Example 9.21

Problem: A circuit has an input *w* and an output *z*. A sequence of pulses is applied on input *w*. The output has to replicate every second pulse, as illustrated in Figure 9.79. Design a suitable circuit.

Figure 9.79 Waveforms for Example 9.21.

Solution: Figure 9.80 shows a possible state diagram and the corresponding flow table. Compare this with the FSM defined in Example 9.4 in Figure 9.13, which specifies a serial parity generator. The only difference is in the output signal. In our case, $z = 1$ only in state *B*. Therefore, the next-state expressions are the same as in Example 9.4. The output expression is

$$z = y_1\bar{y}_2$$



(a) State diagram

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	(A)	B	0
B	C	(B)	1
C	(C)	D	0
D	A	(D)	0

(b) Flow table

Figure 9.80 State diagram and flow table for Example 9.21.

Example 9.22

Problem: Consider the flow table in Figure 9.81. Reduce this flow table and find a state assignment that allows this FSM to be realized as simply as possible, preserving the Moore model. Derive an excitation table.

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	E	C	–	0
B	–	E	H	(B)	1
C	G	–	(C)	F	0
D	A	(D)	–	B	1
E	G	(E)	–	B	0
F	–	D	C	(F)	0
G	(G)	E	C	–	0
H	A	–	(H)	B	1

Figure 9.81 Flow table for Example 9.22.

Solution: Using the partioning procedure on the flow table in Figure 9.81 gives

$$P_1 = (ACEFG)(BDH)$$
$$P_2 = (AG)(B)(C)(D)(E)(F)(H)$$
$$P_3 = P_2$$

Combining A and G produces the flow table in Figure 9.82. A merger diagram for this table is shown in Figure 9.83. Merging the states (A, E), (C, F), and (D, H) leads to the reduced flow table in Figure 9.84. To find a good state assignment, we relabel this flow table as indicated in Figure 9.85, and construct the transition diagram in Figure 9.86*a*. The only problem in this diagram is the transition from state D to state A, labeled as 1. A change from D to A can be made via state C if we specify so in the flow table. Then, a direct

Present state	Next state				Output
	w_2w_1 = 00	01	10	11	z
A	(A)	E	C	–	0
B	–	E	H	(B)	1
C	A	–	(C)	F	0
D	A	(D)	–	B	1
E	A	(E)	–	B	0
F	–	D	C	(F)	0
H	A	–	(H)	B	1

Figure 9.82 Reduction after the partitioning procedure.

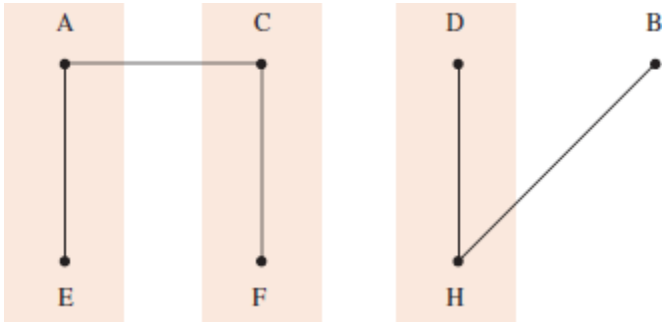


Figure 9.83 Merger diagram for the flow table in Figure 9.82.

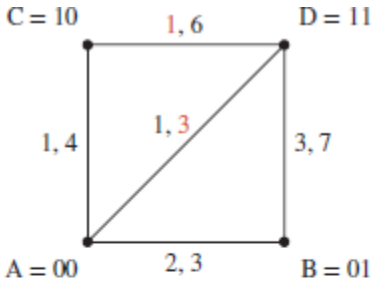
Present state	Next state				Output
	$w_2w_1 =$ 00	01	10	11	z
A	(A)	(A)	C	B	0
B	–	A	D	(B)	1
C	A	D	(C)	(C)	0
D	A	(D)	(D)	B	1

Figure 9.84 Reduced flow table for the FSM in Figure 9.82.

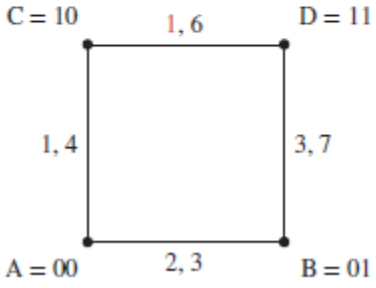
Present state	Next state				Output
	$w_2w_1 =$ 00	01	10	11	z
A	(1)	(2)	4	3	0
B	–	2	7	(3)	1
C	1	6	(4)	(5)	0
D	1	(6)	(7)	3	1

Figure 9.85 Relabeled flow table of Figure 9.84.

transition from *D* to *A* is not needed, as depicted in Figure 9.86*b*. The resulting flow table and the corresponding excitation table are shown in Figure 9.87.



(a) Initial transition diagram



(b) Augmented transition diagram

Figure 9.86 Transition diagrams for Figure 9.85.

Present state	Next state w_2w_1 = 00	01	10	11	Output z
A	(A)	(A)	C	B	0
B	–	A	D	(B)	1
C	A	D	(C)	(C)	0
D	C	(D)	(D)	B	1

(a) Final flow table

Present state y_2y_1	Next state w_2w_1 = 00 Y_2Y_1	01 Y_2Y_1	10 Y_2Y_1	11 Y_2Y_1	Output z
00	(00)	(00)	10	01	0
01	–	00	11	(01)	1
10	00	11	(10)	(10)	0
11	10	(11)	(11)	01	1

(b) Excitation table

Figure 9.87 Excitation and flow tables for Example 9.22.

Example 9.23

Problem: Derive a hazard-free minimum-cost SOP implementation for the function

$$f(x_1, \ldots, x_5) = \sum m(2, 3, 14, 17, 19, 25, 26, 30) + D(10, 23, 27, 31)$$

Solution: The Karnaugh map for the function is given in Figure 9.88. From it, the required expression is derived as

$$f = x_1\bar{x}_3x_5 + x_2x_4\bar{x}_5 + \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_2\bar{x}_3x_4x_5$$

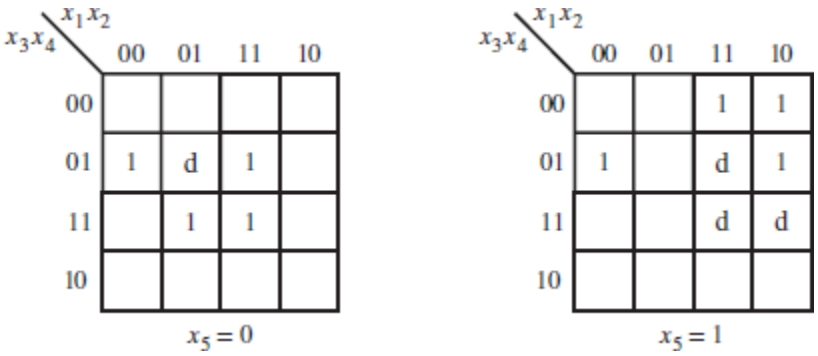


Figure 9.88 Karnaugh map for Example 9.23.

The first three product terms cover all 1s in the map. The fourth term is needed to avoid having a hazard when $x_2x_3x_4x_5 = 0011$ and x_1 changes from 0 to 1 (or 1 to 0). Thus, each pair of adjacent 1s is covered by some prime implicant in the expression.

Problems

Answers to problems marked by an asterisk are given at the back of the book.

***9.1** Derive a flow table that describes the behavior of the circuit in Figure P9.1. Compare your solution with the tables in Figure 9.21.

Is there any similarity?

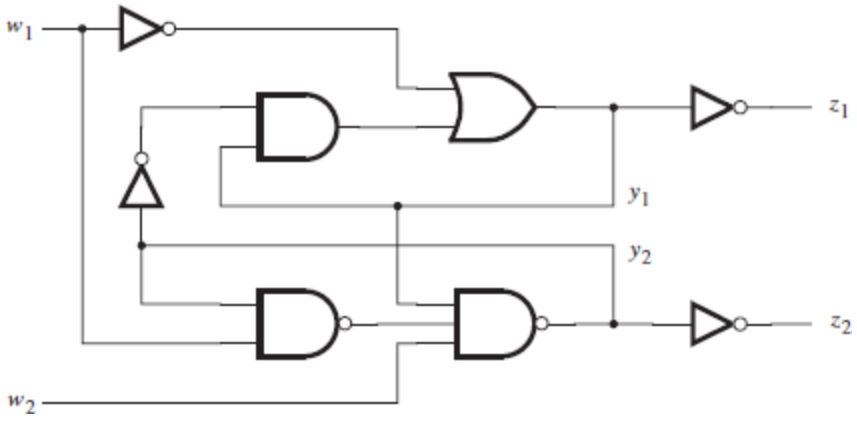


Figure P9.1 Circuit for Problem 9.1.

9.2 Consider the circuit in Figure P9.2. Draw the waveforms for the signals C, z_1, and z_2. Assume that C is a square-wave clock signal and that each gate has a propagation delay Δ. Express the behavior of the circuit in the form of a flow table that would produce the desired signals. (*Hint:* Use the Mealy model.)

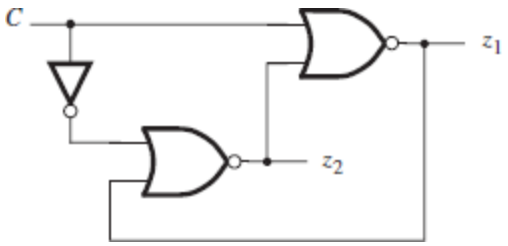


Figure P9.2 Circuit for Problem 9.2.

9.3 Derive the minimal flow table that specifies the same functional behavior as the flow table in Figure P9.3.

Present state	Next state				Output
	w_2w_1 = 00	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	P	–	(C)	–	0
D	(D)	E	F	–	0
E	G	(E)	–	–	0
F	M	–	(F)	–	0
G	(G)	H	I	–	0
H	J	(H)	–	–	0
I	A	–	(I)	–	1
J	(J)	K	L	–	0
K	A	(K)	–	–	1
L	A	–	(L)	–	1
M	(M)	N	O	–	0
N	A	(N)	–	–	1
O	A	–	(O)	–	1
P	(P)	R	S	–	0
R	T	(R)	–	–	0
S	A	–	(S)	–	1
T	(T)	U	V	–	0
U	A	(U)	–	–	1
V	A	–	(V)	–	1

Figure P9.3 Flow table for Problem 9.3.

9.4 Derive the minimal Moore-type flow table that specifies the same functional behavior as the flow table in Figure P9.4.

Present state	Next state				Output z
	$w_2w_1 =$ 00	01	10	11	
A	(A)	B	C	–	0
B	K	(B)	–	H	0
C	F	–	(C)	M	0
D	(D)	E	J	–	1
E	A	(E)	–	M	0
F	(F)	L	J	–	0
G	D	(G)	–	H	0
H	–	G	J	(H)	1
J	F	–	(J)	H	0
K	(K)	L	C	–	1
L	A	(L)	–	H	0
M	–	G	C	(M)	1

Figure P9.4 Flow table for Problem 9.4.

9.5 Find a suitable state assignment using as few states as possible and derive the next-state and output expressions for the flow table in Figure 9.42.

9.6 Find a suitable state assignment for the flow table in Figure 9.42, using pairs of equivalent states, as explained in Example 9.15. Derive the next-state and output expressions.

9.7 Find a state assignment for the flow table in Figure 9.42, using one-hot encoding. Derive the next-state and output expressions.

***9.8** Implement the FSM specified in Figure 9.39, using the merger diagram in Figure 9.40*a*.

9.9 Find a suitable state assignment for the FSM defined by the flow table in Figure P9.5. Derive the next-state and output expressions for the FSM using this state assignment.

Present state	Next state				Output
	w_2w_1 = 00	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	G	0
C	F	–	(C)	G	0
D	(D)	E	C	–	1
E	A	(E)	–	G	0
F	(F)	E	C	–	0
G	–	B	C	(G)	1

Figure P9.5 Flow table for Problem 9.9.

***9.10** Find a hazard-free minimum-cost implementation of the function

$$f(x_1, \ldots, x_4) = \sum m(0, 4, 11, 13, 15) + D(2, 3, 5, 10)$$

9.11 Repeat Problem 9.10 for the function

$$f(x_1, \ldots, x_5) = \sum m(0, 4, 5, 24, 25, 29) + D(8, 13, 16, 21)$$

***9.12** Find a hazard-free minimum-cost POS implementation of the function

$$f(x_1, \ldots, x_4) = \Pi M(0, 2, 3, 7, 10) + D(5, 13, 15)$$

9.13 Repeat Problem 9.12 for the function

$$f(x_1, \ldots, x_5) = \Pi M(2, 6, 7, 25, 28, 29) + D(0, 8, 9, 10, 11, 21, 24, 26, 27, 30)$$

***9.14** Consider the circuit in Figure P9.6. Does this circuit exhibit any hazards?

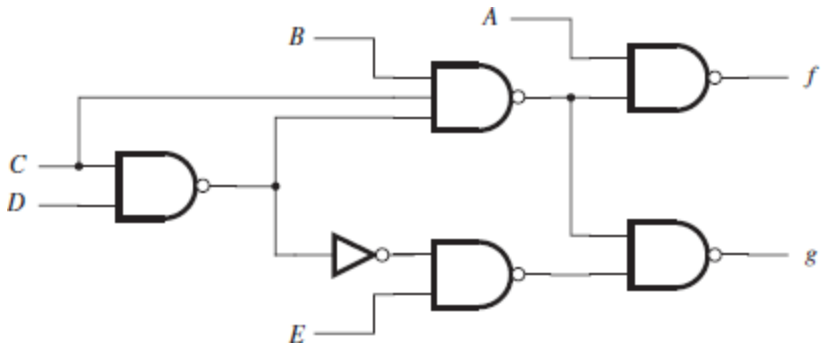


Figure P9.6 Circuit for Problem 9.14.

9.15 Design an original circuit that exhibits a dynamic hazard.

9.16 A control mechanism for a vending machine accepts nickels and dimes. It dispenses merchandise when 20 cents is deposited; it does not give change if 25 cents is deposited. Design the FSM that implements the required control, using as few states as possible. Find a suitable state assignment and derive the next-state and output expressions.

***9.17** Design an asynchronous circuit that meets the following specifications. The circuit has two inputs: a clock input c and a control input w. The output, z, replicates the clock pulses when $w = 1$; otherwise, $z = 0$. The pulses appearing on z must be full pulses. Consequently, if $c = 1$ when w changes from 0 to 1, then the circuit will not produce a partial pulse on z, but will wait until the next clock pulse to generate $z = 1$. If $c = 1$ when w changes from 1 to 0, then a full pulse must be generated; that is, $z = 1$ as long as $c = 1$. Figure P9.7 illustrates the desired operation.

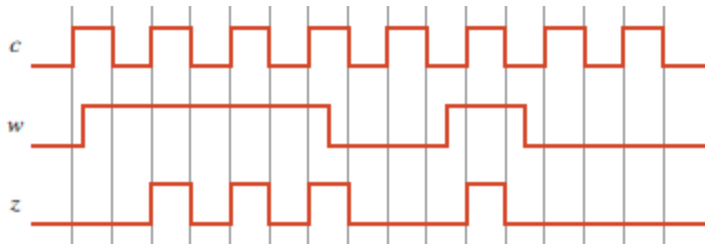


Figure P9.7 Waveforms for Problem 9.17.

9.18 Repeat Problem 9.17 but with the following change in the specification. While $w = 1$, the output z should have only one

pulse; if several pulses occur on *c*, only the first one should be reproduced on *z*.

9.19 Example 9.6 describes a simple arbiter for two devices contending for a shared resource. Design a similar arbiter for three devices that use a shared resource. In case of simultaneous requests, namely, if one device has been granted access to the shared resource and before it releases its request the other two devices make requests of their own, let the priority of the devices be Device 1 > Device 2 > Device 3.

9.20 In the discussion of Example 9.6, we mentioned a possible use of the mutual exclusion element (ME) to prevent both request inputs to the FSM being equal to 1 at the same time. Design an arbiter circuit for this case.

9.21 In Example 9.21 we designed a circuit that replicates every second pulse on input *w* as a pulse on output *z*. Design a similar circuit that replicates every third pulse.

9.22 In Example 9.22 we merged states *D* and *H* to implement the FSM in Figure 9.82. An alternative was to merge states *B* and *H*, according to the merger diagram in Figure 9.83. Derive an implementation using this choice. Derive the resulting excitation table.

chapter

10

COMPUTER AIDED DESIGN TOOLS

CHAPTER OBJECTIVES

In this chapter you will learn how CAD tools can be used to design and implement a logic circuit. The discussion deals with the synthesis and physical design stages in a typical CAD system, including:

- Netlist extraction
- Technology mapping
- Placement
- Routing
- Timing analysis

We introduced CAD tools in Section 2.9, and have discussed them briefly in other chapters. The word *tool* in this context means a software program that allows a user to perform a particular task. In this chapter we describe some of the tools in a typical CAD system in more detail, by showing how a small design example is processed and optimized as it passes through different stages in the CAD flow.

10.1 SYNTHESIS

Figure 10.1, which is reproduced from Figure 2.35, gives an overview of a CAD system. A description of the desired circuit is prepared, usually in the form of a hardware description language like VHDL. The VHDL code is then processed by the synthesis stage of the CAD system. Synthesis is the process of generating a logic circuit from the user's specification. Figure 10.2 shows three typical phases that are found in the synthesis process.

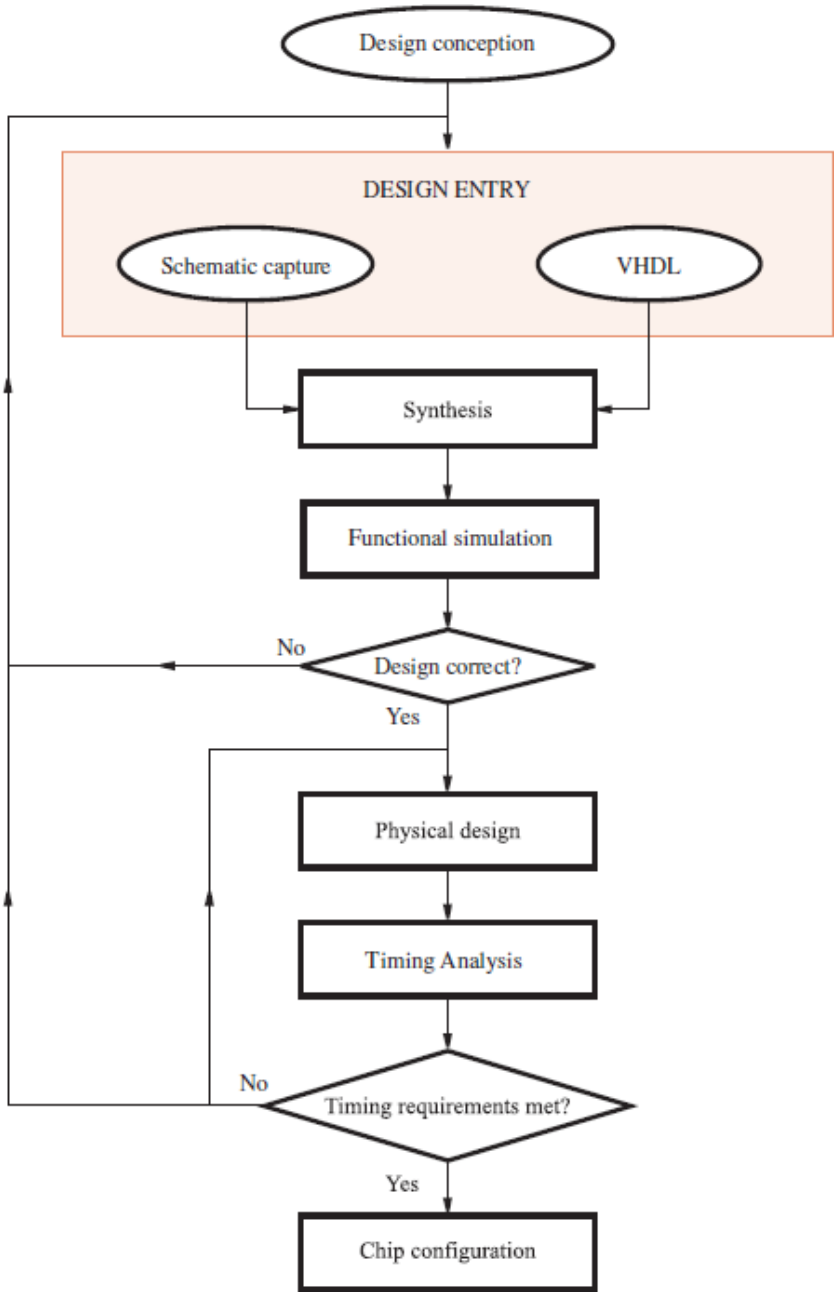


Figure 10.1 A typical CAD system.

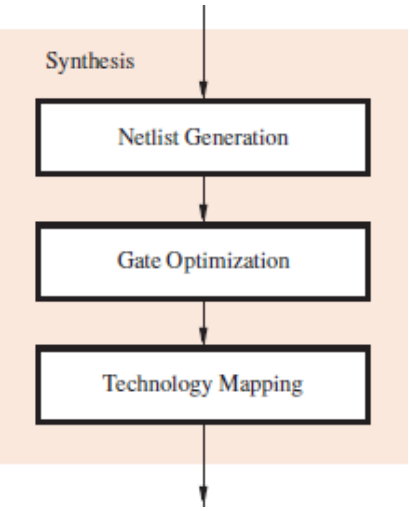


Figure 10.2 The stages included in a synthesis tool.

10.1.1 NETLIST GENERATION

The *netlist generation* phase checks the syntax of the code, and reports any errors such as undefined signals, missing parentheses, and wrong keywords. Once all errors are fixed a circuit netlist is generated as determined by the semantics of the VHDL code. The netlist uses logic expressions to describe the circuit, and includes components such as adders, flip-flops, and finite state machines.

10.1.2 GATE OPTIMIZATION

The next phase is *gate optimization*, which performs the kinds of logic optimizations described in the preceding chapters. These optimizations manipulate the netlist to obtain an equivalent, but better circuit according to the optimization goals. As we said previously, the measurement of what makes one circuit better than another may be based on the cost of the circuit, its speed of operation, or a combination of both.

As an example of results produced by the synthesis phases discussed so far, consider the VHDL code for the *addersubtractor* entity in Figure 10.3, which specifies a circuit that can add or subtract *n*-bit numbers and accumulate the result in a register. From this code the synthesis tool produces a netlist that corresponds to the circuit in Figure 10.4. The input numbers, $A = a_{n-1}, \ldots, a_0$ and $B = b_{n-1}, \ldots, b_0$, are placed into registers *Areg* and *Breg* prior to being used in addition or subtraction operations. These registers synchronize the operation of the circuit if *A* and *B* are externally provided asynchronous inputs. The control input *Sel*

determines the mode of operation. If $Sel = 0$, then *A* is selected as an input to the adder; if $Sel = 1$, then the result register *Zreg* is selected. The control input *AddSub* determines whether the operation is addition or subtraction. The flip-flops in Figure 10.4 for registers *A*, *B*, *Sel*, *AddSub*, and *Overflow* are inferred from the code at the bottom of Figure 10.3*a*. Multiplexers are produced from the *mux2to1* entity, and an adder is generated from the *adderk* entity in Figure 10.3*c*. The exclusive-OR gates connected to register *B*, and the XOR function for the *Overflow* output are generated from the code near the end of the *addersubtractor* entity.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY addersubtractor IS
    GENERIC ( n : INTEGER := 16 ) ;
    PORT ( A, B                    : IN     STD_LOGIC_VECTOR(n–1 DOWNTO 0) ;
           Clock, Reset, Sel, AddSub : IN     STD_LOGIC ;
           Z                       : BUFFER STD_LOGIC_VECTOR(n–1 DOWNTO 0) ;
           Overflow                : OUT    STD_LOGIC ) ;
END addersubtractor ;

ARCHITECTURE Behavior OF addersubtractor IS
    SIGNAL G, H, M, Areg, Breg, Zreg, AddSubR_n : STD_LOGIC_VECTOR(n–1 DOWNTO 0) ;
    SIGNAL SelR, AddSubR, carryout, over_flow : STD_LOGIC ;
    COMPONENT mux2to1
        GENERIC ( k : INTEGER := 8 ) ;
        PORT ( V, W : IN  STD_LOGIC_VECTOR(k–1 DOWNTO 0) ;
               Sel  : IN  STD_LOGIC ;
               F    : OUT STD_LOGIC_VECTOR(k–1 DOWNTO 0) ) ;
    END COMPONENT ;
    COMPONENT adderk
        GENERIC ( k : INTEGER := 8 ) ;
        PORT ( carryin  : IN  STD_LOGIC ;
               X, Y     : IN  STD_LOGIC_VECTOR(k–1 DOWNTO 0) ;
               S        : OUT STD_LOGIC_VECTOR(k–1 DOWNTO 0) ;
               carryout : OUT STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    PROCESS ( Reset, Clock )
    BEGIN
        IF Reset = '1' THEN
            Areg <= (OTHERS => '0'); Breg <= (OTHERS => '0');
            Zreg <= (OTHERS => '0'); SelR <= '0'; AddSubR <= '0'; Overflow <= '0';
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Areg <= A; Breg <= B; Zreg <= M;
            SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
        END IF ;
    END PROCESS ;

. . . continued in Part b
```

Figure 10.3 VHDL code for an accumulator circuit (Part *a*).

```
    nbit_adder: adderk
        GENERIC MAP ( k => n )
        PORT MAP ( AddSubR, G, H, M, carryout ) ;
    multiplexer: mux2to1
        GENERIC MAP ( k => n )
        PORT MAP ( Areg, Z, SelR, G ) ;
    AddSubR_n <= (OTHERS => AddSubR) ;
    H <= Breg XOR AddSubR_n ;
    over_flow <= carryout XOR G(n–1) XOR H(n–1) XOR M(n–1) ;
    Z <= Zreg ;
END Behavior;

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
GENERIC ( k : INTEGER := 8 ) ;
    PORT ( V, W  : IN  STD_LOGIC_VECTOR(k–1 DOWNTO 0) ;
           Sel   : IN  STD_LOGIC ;
           F     : OUT STD_LOGIC_VECTOR(k–1 DOWNTO 0) );
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( V, W, Sel )
    BEGIN
        IF Sel = '0' THEN
            F <= V ;
        ELSE
            F <= W ;
        END IF ;
    END PROCESS ;
END Behavior ;

. . . continued in Part c
```

Figure 10.3 VHDL code for an accumulator circuit (Part *b*).

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adderk IS
    GENERIC ( k : INTEGER := 8 ) ;
    PORT ( carryin  : IN  STD_LOGIC ;
           X, Y     : IN  STD_LOGIC_VECTOR(k–1 DOWNTO 0) ;
           S        : OUT STD_LOGIC_VECTOR(k–1 DOWNTO 0) ;
           carryout : OUT STD_LOGIC ) ;
END adderk ;

ARCHITECTURE Behavior OF adderk IS
    SIGNAL Sum : STD_LOGIC_VECTOR(k DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + Y + carryin ;
    S <= Sum(k–1 DOWNTO 0) ;
    carryout <= Sum(k) ;
END Behavior ;
```

Figure 10.3 VHDL code for an accumulator circuit (Part *c*).

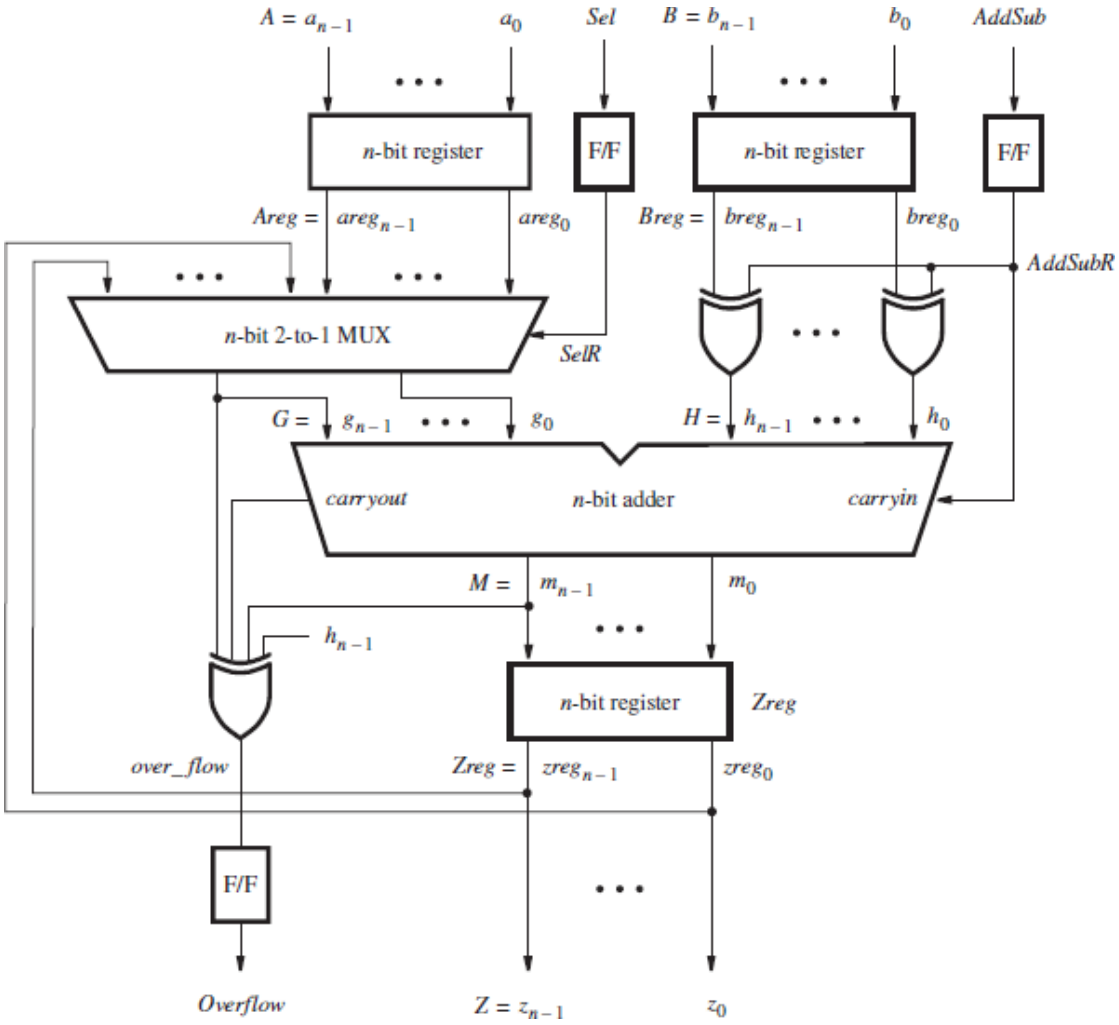


Figure 10.4 Circuit specified by the code in Figure 10.3.

10.1.3 TECHNOLOGY MAPPING

The final phase of synthesis is *technology mapping*. This phase determines how each component in the netlist can be realized in the resources available in the target chip, such as an FPGA. A logic block for a typical FPGA is depicted in Figure 10.5*a*. It contains a four-input lookup-table (LUT) and a flip-flop, and has two outputs. A multiplexer is provided to allow loading of the flip-flop from the LUT or directly from input *In3*. Another multiplexer allows the stored value in the flip-flop to be fed back to one input of the LUT. There are a number of different ways, or *modes*, in which this logic block can be used. The most straightforward choice is to implement a function of up to four inputs in the LUT, and store this function's value in the flip-flop; both the LUT and flip-flop can provide outputs from the logic block. Parts (*b*) to (*e*) of the figure illustrate four other modes of using the block. In parts (*b*) and (*c*) only the LUT or the flip-flop

is used, but not both. In part (*d*) only the LUT provides an output of the logic block, and one of the LUT's inputs is connected to the flip-flop.

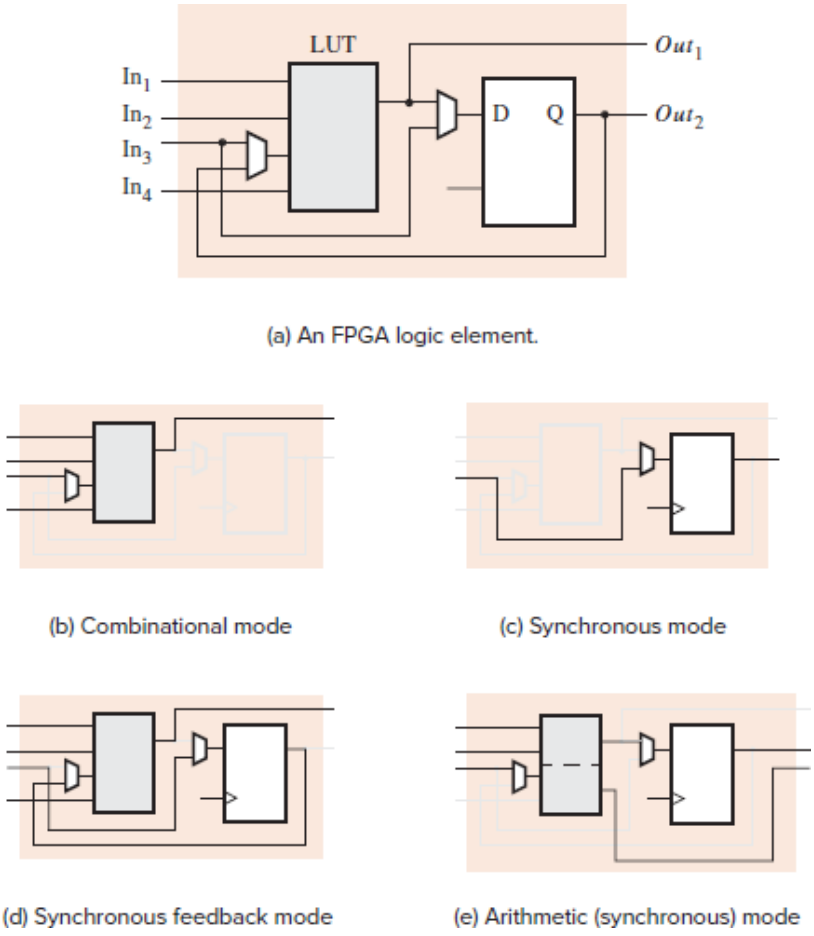


Figure 10.5 Different modes of an FPGA logic block.

FPGAs often contain dedicated circuitry for implementation of adder circuits. Figure 10.5*e* shows one way in which such circuitry can be realized. The LUT is used in two halves, where one half produces the sum function of three LUT inputs and the other half produces the carry function of these inputs. The sum function can provide an output of the block or be stored in the flip-flop, and the carry function provides a special output signal. This carry output connects directly to a neighboring logic block that uses it as a carry input. This block in turn generates the next stage of carry output, and so on. In this way, direct connections between neighboring logic blocks are used to form carry chains.

Figure 10.6 shows a part of the results of technology mapping for the netlist generated for Figure 10.4. Each logic block is highlighted with a red square, and has a label on the lower left corner that indicates which mode in Figure 10.5 is being used. The figure shows bit h_0 from Figure 10.4, which is produced by a

logic block in mode *d*. This block uses a flip-flop to store the value of primary input b_0, and implements an XOR function in its LUT, which is needed in subtraction operations to complement the number *B*. One input of the XOR is provided by the logic block in mode *c* that stores in a flip-flop the value of the *AddSub* input. This flip-flop also drives 15 other logic blocks that implement h_1, …, h_{15}, but these blocks are not shown in the figure.

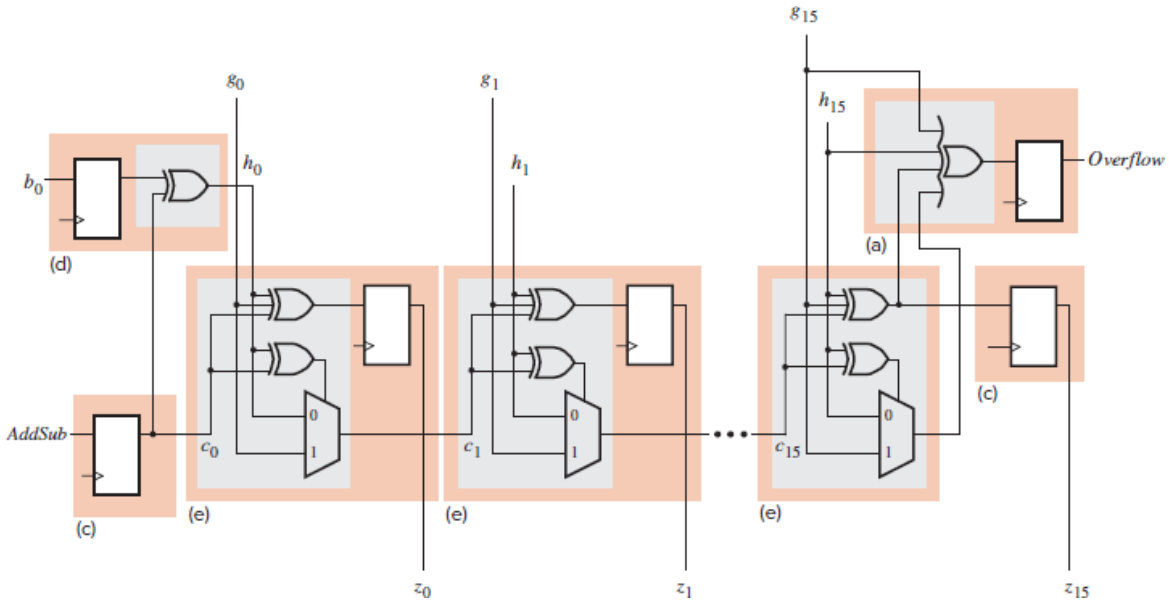


Figure 10.6 A part of the circuit in Figure 10.4 after technology mapping.

The *AddSub* flip-flop is connected to the carry-in of the first logic block in the adder. This block uses mode *e* to produce sum and carry outputs. The sum is stored in a flip-flop that produces z_0, and the carry feeds the next stage of the adder. The figure shows the carry function in the form

$$c_1 = (\overline{c_0 \oplus h_0}) \cdot h_0 + (c_0 \oplus h_0) \cdot g_0$$

This expression is functionally equivalent to the one used in Chapter 3, which has the form $c_1 = c_0h_0 + c_0g_0 + h_0g_0$, but it represents more closely how the carry chain is built in an FPGA. The last logic block of the adder in Figure 10.6 does not use its flip-flop, because the sum output has to be connected directly to the logic block that implements the *Overflow* signal. Although not shown in Figure 10.5, we assume that the sum output can be provided in either combinational or registered form, but not both; so a separate logic block in mode *c* is needed for the z_{15} signal.

Figure 10.6 shows only a few of the logic blocks that a technology mapping tool would create for implementing our circuit. In general, there are many

different ways in which technology mapping can be done, and each method will lead to equivalent, but different circuits. The reader can consult references [1–3] for a detailed discussion of technology mapping approaches.

10.2 Physical Design

The next stages following synthesis in Figure 10.1 are functional simulation and physical design. As we said in Section 2.9, functional simulation involves applying test patterns to the synthesized netlist and checking to see if it produces the correct outputs. The simulation assumes that there are no propagation delays in the circuit, because the intent is to evaluate the basic functionality rather than timing. The netlist used by a functional simulator could be either the version before technology mapping, or after.

Once the netlist produced by synthesis is functionally correct, the physical design stage can be performed. This stage determines exactly how the synthesized netlist will be implemented in the target chip. As illustrated in Figure 10.7, three phases are involved: placement, routing, and timing analysis.

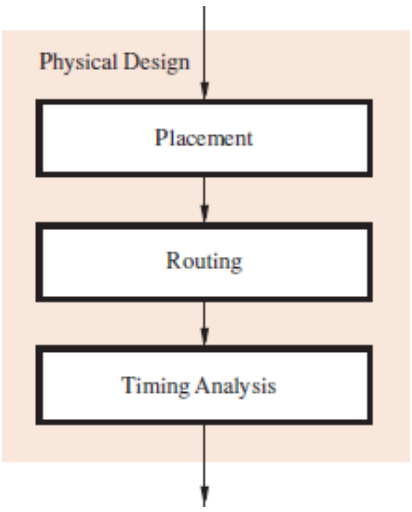


Figure 10.7 Phases in physical design.

10.2.1 Placement

The placement phase chooses a location in the target device for each logic block in the technology-mapped netlist. An example of a placement result is given in Figure 10.8. It shows an array of logic blocks in a small portion of an FPGA chip. The white squares represent unoccupied blocks and the gray squares show the

placement of blocks that implement the circuit of Figure 10.4. There is a total of 53 logic blocks in this circuit, including the ones shown in Figure 10.6. Also shown in Figure 10.8 is the placement of some of the primary inputs to the circuit, which are assigned to pins around the chip periphery.

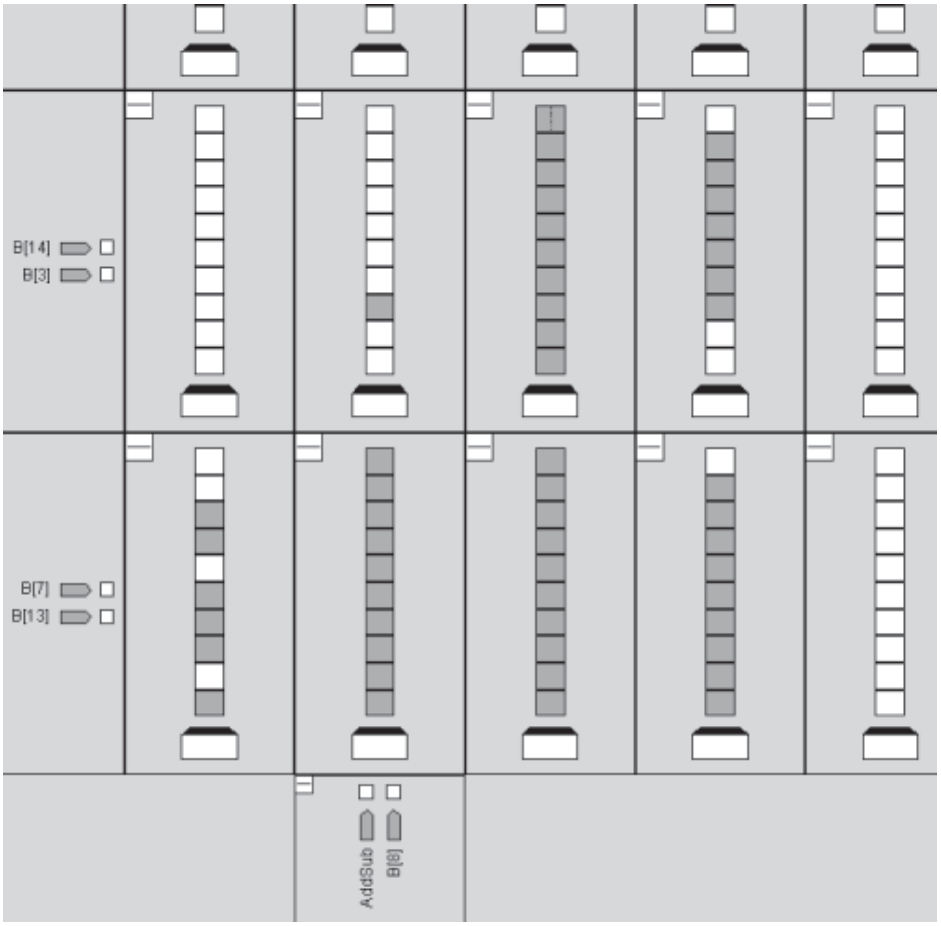


Figure 10.8 Placement of the circuit in Figure 10.6.

To find a good placement solution a number of different locations have to be considered for each logic block. For a large circuit, which may contain tens of thousands of blocks, this is a hard problem to solve. To appreciate the complexity involved, consider how many different placement solutions are possible for a given circuit. Assume that the circuit has N logic blocks, and it is to be placed in an FPGA that also contains exactly N blocks. A placement tool has N choices for the location of the first block that it selects. There remain N-1 choices for the second block, N-2 choices for the third, and so on. Multiplying these choices gives a total of $(N)(N\text{-}1)\cdots(1) = N!$ possible placement solutions. For even moderate values, $N!$ is a huge number, which means that heuristic techniques must be used to find a good solution while considering only a small fraction of

the total number of choices. A typical commercial placement tool operates by constructing an initial placement configuration and then moving logic blocks to different locations in an iterative manner. For each iteration the quality of the solution is assessed by using metrics that estimate the speed of operation of the implemented circuit, or its cost. The placement problem has been studied extensively and is described in detail in references [4–7].

10.2.2 ROUTING

Once a location in the chip is chosen for each logic block in a circuit, the routing phase connects the blocks together by using the wires that exist in the chip. An example of a routing solution for the placement in Figure 10.8 is given in Figure 10.9. In addition to showing the logic blocks, this figure also displays some of the wires in the chip. Wires that are being used by the implemented circuit are shaded in gray. The figure depicts both individual wires, which may be of various lengths, and bundles of wires, which are shown as gray rectangles. The routing CAD tool tries to make the best use of various kinds of wires, such as efficient connections for carry chains. Figure 10.9 shows an example of the carry chain path from Figure 10.6. Black lines highlight the carry chain wires, which connect through the stages of the adder, ending at the *Overflow* register. A detailed discussion of routing tools can be found in references [3], [5–6], and [8].

Parameter	Actual	Required	Slack	From	To
F_{max}	261.1 MHz	200 MHz	1.17 ns	*AddSub*	*Overflow*
t_{su}	2.356 ns	10.0 ns	7.644 ns	b_0	$breg_0$
t_{co}	6.772 ns	10.0 ns	3.228 ns	$breg_0$	z_0
t_h	0.240 ns	N/A	N/A	b_1	$breg_1$

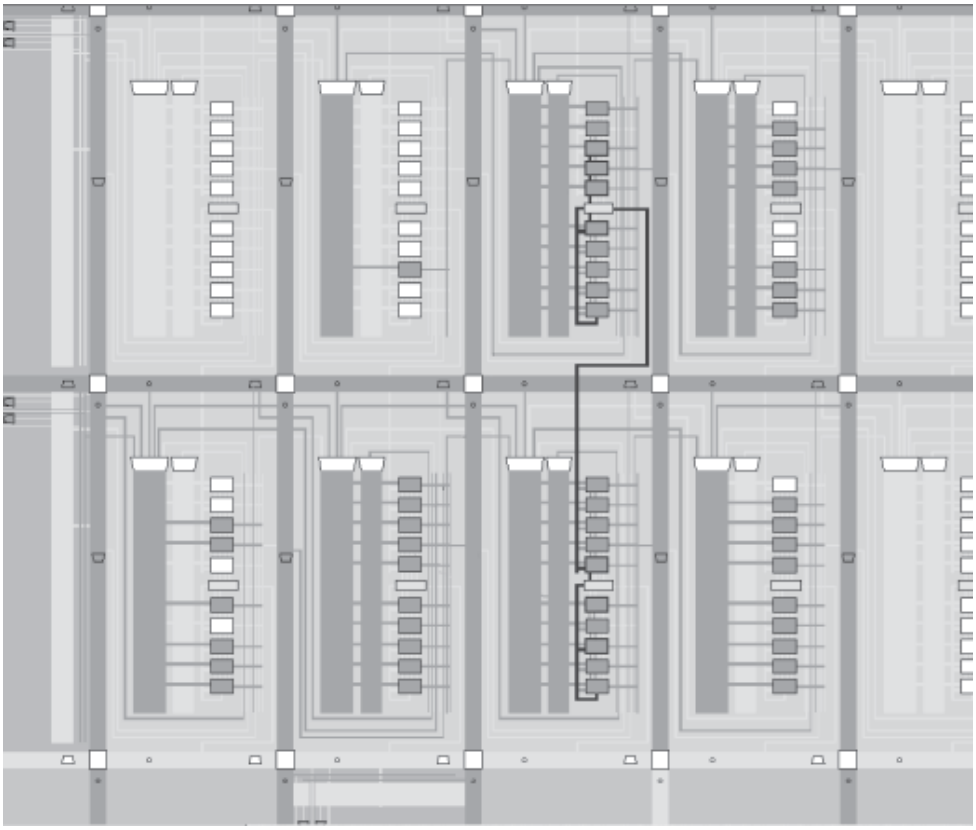


Figure 10.9 Routing for the placement in Figure 10.8.

10.2.3 TIMING ANALYSIS

After routing is complete the timing delays for the implemented circuit are known, because the CAD system computes the timing delays of all blocks and wires in the chip. A timing analysis tool examines this delay information and produces a set of tables that quantify the circuit's performance. An example of a timing analysis result is given in Table 10.1, which lists four parameters: F_{max}, t_{su}, t_{co}, and t_h. The F_{max} value specifies the maximum operating frequency of the circuit's *clock*. This value is determined by the path with the longest propagation delay, often called the *critical path*, between any two flip-flops in the circuit. As shown in Sections 5.15 and 7.8, the path delay must account for the delays through logic blocks and wires, as well as the flip-flop clock-to-Q delay (t_{cQ}) and setup (t_{su}) parameters. In our example the critical path delay is $1/261.1 \times 10^6 = 3.83$ ns ns. The last two columns in the F_{max} row show that the path starts at the *AddSub* flip-flop and ends at the *Overflow* flip-flop shown in Figure 10.6.

Table 10-1 A summary of timing analysis results.

Most CAD systems allow users to specify the timing constraints for their circuit. In Table 10.1 we assume the user has specified that the circuit *clock* has to operate correctly up to a frequency of 200 MHz. The difference between this requirement and the result that is obtained by the CAD tools is referred to as *slack*. In the table, the requirement is that the propagation delays must not exceed $1/200 \times 10^6 = 5$ ns; the result is 3.83 ns, which gives a slack value of 1.17 ns. This positive slack means that the constraints have been met with some room to spare. If the obtained result had a negative slack, then the user's requirements would not have been met, and it would be necessary to modify the VHDL code or settings used in the CAD tool to try to meet the constraints.

The other rows in Table 10.1 show the timing results for the design's primary inputs and outputs. The t_{su} result indicates the worst-case setup requirement is 2.356 ns, from pin b_0 to flip-flop $breg_0$. This parameter means that the b_0 signal must have a stable value at least 2.356 ns before each active edge of the clock signal at its assigned pin. Since the designer specified a worst-case setup constraint of 10 ns, the obtained result means that the implemented circuit exceeds (meets) the requirement by a slack value of 7.644 ns. The worst-case clock-to-output delay for our circuit is 6.772 ns, from flip-flop $zreg_0$ to pin z_0. This means that the propagation delay from an active edge of the clock signal at its pin to a corresponding change in the z_0 signal at its pin is 6.772 ns. Since the designer's constraint specifies that a 10 ns t_{co} is allowed, the available slack is 3.228 ns.

The last row in Table 10.1 gives a maximum hold time of 0.24 ns, for the path from pin b_1 to flip-flop $breg_1$. Hence, the signal at pin b_1 must maintain a stable value for at least 0.24 ns after each active edge at the clock pin. We assume that no constraint was set for this parameter, thus no slack value is shown.

Table 10.1 lists only the worst-case paths for F_{max}, t_{su}, t_{co}, and t_h. The implemented circuit will have a number of other paths that have smaller delays and greater slack values. A timing analysis tool typically provides additional tables for each parameter, which list more paths.

10.3 Concluding Remarks

In this chapter we explained briefly a typical design flow made possible by the existence of powerful CAD tools. We considered only the most important subset of the tools available in commercial CAD systems. To learn more the reader can consult the literature or visit the web sites of CAD tool suppliers. Some of the

major vendors of CAD tools include: Cadence Design Systems, Mentor Graphics, Microchip, Synopsys, Intel, and Xilinx.

References

1. R. Murgai, R. Brayton, A. Sangiovanni-Vincentelli, *Logic Synthesis for FieldProgrammable Gate Arrays* (Kluwer Academic Publishers, 1995).
2. J. Cong and Y. Ding, *FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs* (in IEEE Transactions on Computer-aided Design 13 (1), January 1994).
3. S. Brown, R. Francis, J. Rose, Z. Vranesic, *Field-Programmable Gate Arrays*, (Kluwer Academic Publishers, 1995).
4. M. Breuer, *A Class of Min-cut Placement Algorithms*, (in Design Automation Conference, pages 284–290, IEEE/ACM, 1977).
5. Carl Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*, (Kluwer Academic Publishers, 1988).
6. V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, (Kluwer Academic Publishers, 1999).
7. M. Sarrafzadeh, M. Wang, and X. Yang, *Modern Placement Techniques*, (Kluwer Academic Publishers, 2003).
8. L. McMurchie and C. Ebeling, *PathFinder: A Negotiation-Based PerformanceDriven Router for FPGAs*, (in International Symposium on Field Programmable Gate Arrays, Monterey, Ca., Feb. 1995).

chapter

11

TESTING OF LOGIC CIRCUITS

CHAPTER OBJECTIVES

In this chapter you will be introduced to:

- Various techniques for testing of digital circuits
- Representation of typical faults in a circuit
- Derivation of tests needed to test the behavior of a circuit
- Design of circuits for easy testability

In the previous chapters we have discussed the design of logic circuits. Following a sound design procedure, we expect that the designed circuit will perform as required. But how does one verify that the final circuit indeed achieves the design objectives? It is essential to ascertain that the circuit exhibits the required functional behavior and that it meets any timing constraints that are imposed on the design. We have discussed the timing issues in several places in the book. In this chapter we will discuss some testing techniques that can be used to verify the functionality of a given circuit.

There are several reasons for testing a logic circuit. When the circuit is first developed, it is necessary to verify that the designed circuit meets the required functional and timing specifications. When multiple copies of a correctly designed circuit are being manufactured, it is essential to test each copy to ensure

that the manufacturing process has not introduced any flaws. It is also necessary to test circuits used in equipment that is installed in the field when it is suspected that there may be something wrong.

The basis of all testing techniques is to apply predefined sets of inputs, called *tests*, to a circuit and compare the outputs observed with the patterns that a correctly functioning circuit is supposed to produce. The challenge is to derive a relatively small number of tests that provide an adequate indication that the circuit is correct. The exhaustive approach of applying all possible tests is impractical for large circuits because there are too many possible tests.

11.1 Fault Model

A circuit functions incorrectly when there is something wrong with it, such as a transistor fault or an interconnection wiring fault. Many things can go wrong, leading to a variety of faults. A transistor switch can break so that it is permanently either closed or open. A wire in the circuit can be shorted to V_{DD} or to ground, or it can be simply broken. There can be an unwanted connection between two wires. A logic gate may generate a wrong output signal because of a fault in the circuitry that implements the gate. Dealing with many different types of faults is cumbersome. Fortunately, it is possible to restrict the testing process to some simple faults, and obtain generally satisfactory results.

11.1.1 Stuck-at Model

Most circuits discussed in this text use logic gates as the basic building blocks. A good model for representing faults in such circuits is to assume that all faults manifest themselves as some wires (inputs or outputs of gates) being permanently stuck at logic value 0 or 1. We indicate that a wire, *w*, has an undesirable signal that always corresponds to the logic value 0 by saying that *w* is *stuck-at-0*, which is denoted as *w*/0. If *w* has an undesirable signal that is always equal to logic 1, then *w* is *stuck-at-1*, which is denoted as *w*/1.

An obvious example of a stuck-at fault is when an input to a gate is incorrectly connected to a power supply, either V_{DD} or ground. But the stuck-at model is also useful for dealing with faults of other types, which often cause the same problems as if a wire were stuck at a particular logic value. The exact impact of a fault in the circuitry that implements a logic gate depends on the particular technology used. We will restrict our attention to the stuck-at faults and will examine the testing process assuming that these are the only faults that can occur.

11.1.2 SINGLE AND MULTIPLE FAULTS

A circuit can be faulty because it has either a single fault or possibly many faults. Dealing with multiple faults is difficult because each fault can occur in many different ways. A pragmatic approach is to consider single faults only. Practice has shown that a set of tests that can detect all single faults in a given circuit can also detect the vast majority of multiple faults.

A fault is detected if the output value produced by the faulty circuit is different from the value produced by the good circuit when an appropriate test is applied as input. Each test is supposed to be able to detect the occurrence of one or more faults. A complete set of tests used for a given circuit is referred to as the *test set*.

11.1-3 CMOS CIRCUITS

CMOS logic circuits present some special problems in terms of faulty behavior. The transistors may fail in permanently open or shorted (closed) state. Many such failures manifest themselves as stuck-at faults. But some produce entirely different behavior. For example, transistors that fail in the shorted state may cause a continuous flow of current from V_{DD} to ground, which can create an intermediate output voltage that may not be determined as either logic 0 or 1. Transistors failing in the open state may lead to conditions where the output capacitor retains its charge level because the switch that is supposed to discharge it is broken. The effect is that a combinational CMOS circuit starts behaving as a sequential circuit.

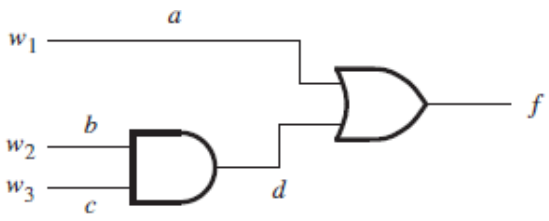
Specific techniques for testing of CMOS circuits are beyond the scope of this book. We will assume that a test set developed using the stuck-at model will provide an adequate coverage of faults in all circuits.

11.2 COMPLEXITY OF A TEST SET

There is large difference in testing combinational and sequential circuits. Combinational circuits can be tested adequately regardless of their design. Sequential circuits present a much greater challenge because the behavior of a circuit under test is influenced not only by the tests that are applied to the external inputs but also by the states that the circuit is in when the tests are applied. It is very difficult to test a sequential circuit designed by a designer who does not take its testability into account. However, it is possible to design such circuits to make them more easily testable, as we will discuss in Section 11.6. We will start by considering the testing of combinational circuits.

An obvious way to test a combinational circuit is to apply a test set that comprises all possible input valuations. Then it is only necessary to check if the output values produced by the circuit are the same as specified in a truth table that defines the circuit. This approach works well for small circuits, where the test set is not large, but it becomes totally impractical for large circuits with many input variables. Fortunately, it is not necessary to apply all 2^n valuations as tests for an n-input circuit. A complete test set, capable of detecting all single faults, usually comprises a much smaller number of tests.

Figure 11.1*a* gives a simple three-input circuit for which we want to determine the smallest test set. An exhaustive test set would include all eight input valuations. This circuit involves five wires, labeled *a*, *b*, *c*, *d*, and *f* in the figure. Using our fault model, each wire can be stuck either at 0 or 1.



(a) Circuit

Test $w_1w_2w_3$	Fault detected									
	$a/0$	$a/1$	$b/0$	$b/1$	$c/0$	$c/1$	$d/0$	$d/1$	$f/0$	$f/1$
000		√						√		√
001		√		√				√		√
010		√				√		√		√
011			√		√		√		√	
100	√								√	
101	√								√	
110	√								√	
111									√	

(b) Faults detected by the various input valuations

Figure 11.1 Fault detection in a simple circuit.

Figure 11.1*b* enumerates the utility of the eight input valuations as possible tests for the circuit. The valuation $w_1w_2w_3 = 000$ can detect the occurrence of a stuck-at-1 fault on wires *a*, *d*, and *f*. In a good circuit this test results in the output $f = 0$. However, if any of the faults $a/1$, $d/1$, or $f/1$ occurs, then the circuit will

produce $f = 1$ when the input valuation 000 is applied. The test 001 causes $f = 0$ in the good circuit, and it results in $f = 1$ if any of the faults $a/1$, $d/1$, or $f/1$ occurs. This test can detect the occurrence of four different faults. We say that it *covers* these faults. The last test, 111, can detect only one fault, $f/0$.

A minimal test set that covers all faults in the circuit can be derived from the table by inspection. Some faults are covered by only one test, which means that these tests must be included in the test set. The fault $b/1$ is covered only by 001. The fault $c/1$ is covered only by 010. The faults $b/0$, $c/0$, and $d/0$ are covered only by 011. Therefore, these three tests are essential. For the remaining faults there is a choice of tests that can be used. Selecting the tests 001, 010, and 011 covers all faults except $a/0$. This fault can be covered by three different tests. Choosing 100 arbitrarily, a complete test set for the circuit is

$$\text{Test set} = \{001, 010, 011, 100\}$$

The conclusion is that all possible stuck-at faults in this circuit can be detected using four tests, rather than the eight tests that would be used if we simply tried to test the circuit using its complete truth table.

The size of the complete test set for a given n-input circuit is generally much smaller than 2^n. But this size may still be unacceptably large in practical terms. Moreover, deriving the minimal test set is likely to be a daunting task for even moderately sized circuits. Certainly, the simple approach of Figure 11.1 is not practical. In the next section we will explore a more interesting approach.

11.3 Path Sensitizing

Deriving a test set by considering the individual faults on all wires in a circuit, as done in Section 11.2, is not attractive from the practical point of view. There are too many wires and too many faults to consider. A better alternative is to deal with several wires that form a path as an entity that can be tested for several faults using a single test. It is possible to activate a path so that the changes in the signal that propagates along the path have a direct impact on the output signal.

Figure 11.2 illustrates a path from input w_1 to output f, through three gates, which consists of wires a, b, c, and f. The path is activated by ensuring that other paths in the circuit do not determine the value of the output f. Thus the input w_2 must be set to 1 so that the signal at b depends only on the value at a. The input w_3 must be 0 so that it does not affect the NOR gate, and w_4 must be 1 to not affect the AND gate. Then if $w_1 = 0$ the output will be $f = 1$, whereas $w_1 = 1$ will

cause $f = 0$. Instead of saying that the path from w_1 to f is activated, a more specific term is used in technical literature, which says that the path is *sensitized*.

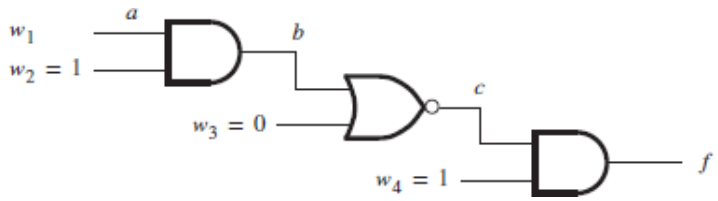


Figure 11.2 A sensitized path.

To sensitize a path through an input of an AND or NAND gate, all other inputs must be set to 1. To sensitize a path through an input of an OR or NOR gate, all other inputs must be 0.

Consider now the effect of faults along a sensitized path. The fault $a/0$ in Figure 11.2 will cause $f = 1$ even if $w_1 = 1$. The same effect occurs if the faults $b/0$ or $c/1$ are present. Thus the test $w_1\ w_2\ w_3\ w_4 = 1101$ detects the occurrence of faults $a/0$, $b/0$, and $c/1$. Similarly, if $w_1 = 0$, the output should be $f = 1$. But if any of the faults $a/1$, $b/1$, or $c/0$ is present, the output will be $f = 0$. Hence these three faults are detectable using the test 0101. The presence of any stuck-at fault along the sensitized path is detectable by applying only two tests.

The number of paths in a given circuit is likely to be much smaller than the number of individual wires. This suggests that it may be attractive to derive a test set based on the sensitized paths. This possibility is illustrated in the next example.

Example 11.1

PATH-SENSITIZED TESTS Consider the circuit in Figure 11.3. This circuit has five paths. The path w _1 - c - f is sensitized by setting $w_2 = 1$ and $w_4 = 0$. It doesn't matter whether w_3 is 0 or 1, because $w_2 = 1$ causes the signal on wire b to be equal to 0, which forces $d = 0$ regardless of the value of w_3. Thus the path is sensitized by setting $w_2\ w_3\ w_4 = 1x0$, where the symbol x means that the value of w_3 does not matter. Now the tests $w_1\ w_2\ w_3\ w_4 = 01x0$ and 11x0 detect all faults along this path. The second path, w_2 - c - f, is tested using 1000 and 1100. The path w_2 - b - d - f is tested using 0010 and 0110. The tests for the path w_3 - d - f are x000 and x010. The fifth path, w_4 - f, is tested with 0x00 and 0x01. Instead of using all ten of these tests, we can observe that the test 0110 serves also as the test

01x0, the test 1100 serves also as 11x0, the test 1000 serves also as x000, and the test 0010 serves also as x010. Therefore, the complete test set is

Test set = {0110, 1100, 1000, 0010, 0x00, 0x01}

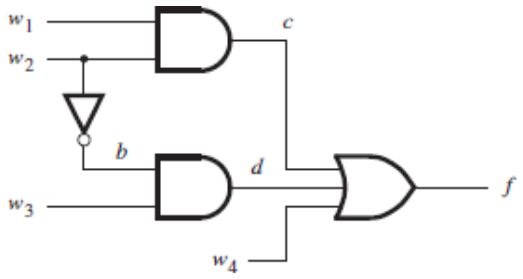


Figure 11.3 Circuit for Example 11.1.

While this approach is simpler, it is still impractical for large circuits. But the concept of path sensitizing is very useful, as we will see in the discussion that follows.

11.3.1 DETECTION OF A SPECIFIC FAULT

Suppose that we suspect that the circuit in Figure 11.3 has a fault where the wire b is stuck-at-1. A test that determines the presence of this fault can be obtained by sensitizing a path that propagates the effect of the fault to the output, f, where it can be observed. The path goes from b to d to f. It is necessary to set $w_3 = 1$, $w_4 = 0$, and $c = 0$. The latter can be accomplished by setting $w_1 = 0$. If b is stuck-at-1, then it is necessary to apply an input that would normally produce the value of 0 on the wire b, so that the output values in good and faulty circuits would be different. Hence w_2 must be set to 1. Therefore, the test that detects the b/1 fault is $w_1\ w_2\ w_3\ w_4 = 0110$.

In general, the fault on a given wire can be detected by propagating the effect of the fault to the output, sensitizing an appropriate path. This involves assigning values to other inputs of the gates along the path. These values must be obtainable by assigning specific values to the primary inputs, which may not always be possible. Example 11.2 illustrates the process.

Example 11.2

Fault Propagation As the effect of a fault propagates through the gates along a sensitized path, the polarity of the signal will change when passing through an

inverting gate. Let the symbol D denote a stuck-at-0 fault in general. The effect of the stuck-at-0 fault will be unaltered when passed through an AND or OR gate. If D is on an input of an AND (OR) gate and the other inputs are set to 1 (0), then the output of the gate will behave as having D on it. But if D is on an input of a NOT, NAND, or NOR gate, then the output will appear to be stuck-at-1, which is denoted as $\overline{D}$.

Figure 11.4 shows how the effect of a fault can be propagated using the D and $\overline{D}$ symbols. Suppose first that there is a stuck-at-0 fault on wire b; that is, $b/0$. We want to propagate the effect of this fault along the path b - h - f. This can be done as indicated in Figure 11.4*b*. Setting $g = 1$ propagates the fault to the wire h. Then h appears to be stuck-at-1, denoted by $\overline{D}$. Next the effect is propagated to f by setting $k = 1$. Since the last NAND also inverts the signal, the output becomes equal to D, which is equivalent to $f/0$. Thus in a good circuit the output should be 1, but in a faulty circuit it will be 0. Next we must ascertain that it is possible to have $g = 1$ and $k = 1$ by assigning the appropriate values to the primary input variables. This is called the *consistency check*. By setting $c = 0$, both g and k will be forced to 1, which can be achieved with $w_3 = w_4 = 1$. Finally, to cause the propagation of the fault D on wire b, it is necessary to apply a signal that causes b to have the value 1, which means that either w_1 or w_2 has to be 0. Then the test $w_1 w_2 w_3 w_4 = 0011$ detects the fault $b/0$.

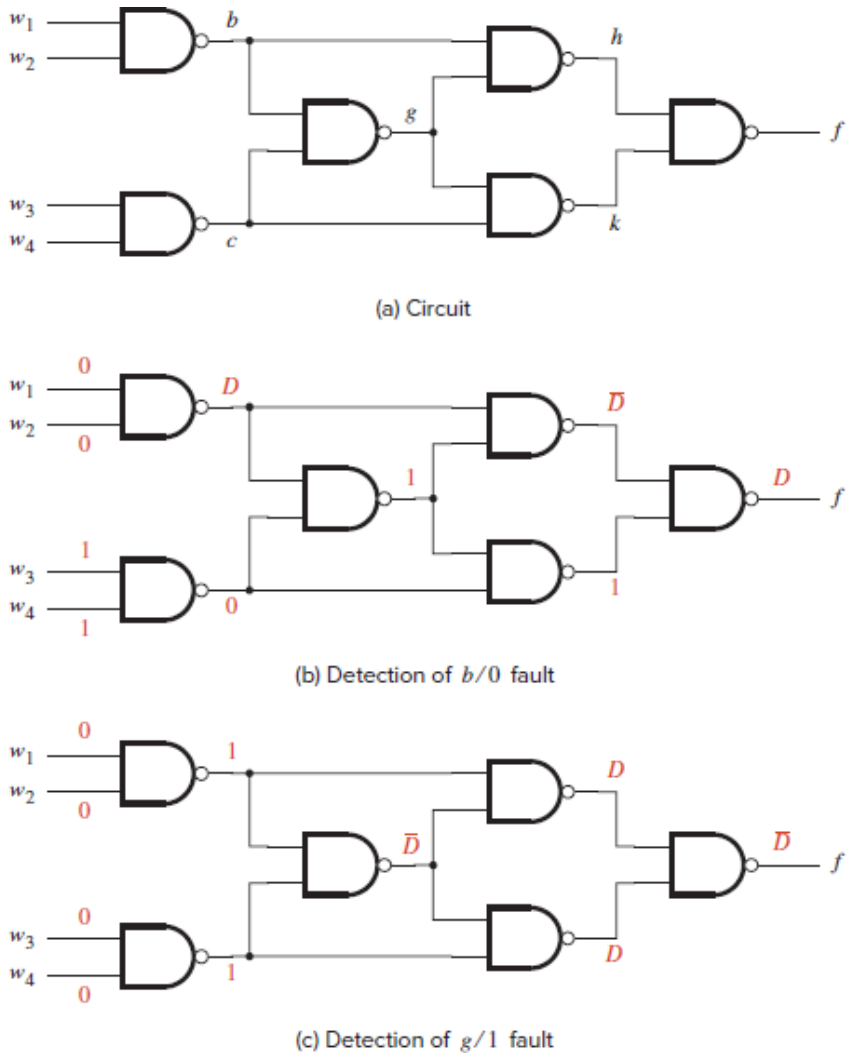


Figure 11.4 Detection of faults.

Suppose next that the wire g is stuck-at-1, denoted by $\bar{D}$. We can try to propagate the effect of this fault through the path g - h - f by setting $b = 1$ and $k = 1$. To make $b = 1$, we set $w_1 = w_2 = 0$. To make $k = 1$, we have to make $c = 0$. But it is also necessary to cause the propagation of the $\bar{D}$ fault on g by means of a signal that makes $g = 0$ in the good circuit. This can be done only if $b = c = 1$. The problem is that at the same time we need $c = 0$, to make $k = 1$. Therefore, the consistency check fails, and the fault $g/1$ cannot be propagated in this way.

Another possibility is to propagate the effect of the fault along two paths simultaneously, as shown in Figure 11.4*c*. In this case the fault is propagated along the paths g - h - f and g - k - f. This requires setting $b = 1$ and $c = 1$, which also happens to be the condition needed to cause the propagation as explained above. The test 0000 achieves the desired objective of detecting $g/1$. Observe that

if D (or $\bar{D}$) appears on both inputs of a NAND gate, the output value will be $\bar{D}$ (or D).

The idea of propagating the effect of faults using path sensitizing has been exploited in a number of methods for derivation of test sets for fault detection. The scheme illustrated in Figure 11.4 indicates the essence of the *D*-algorithm, which was one of the first practical schemes developed for fault detection purposes [1]. Other techniques have grown from this basic approach.

11.4 Circuits with Tree Structure

Circuits with a treelike structure, where each gate has a fan-out of 1, are particularly easy to test. The most common forms of such circuits are the sum-of-products or product-of-sums. Since there is a unique path from each primary input to the output of the circuit, it is sufficient to derive the tests for faults on the primary inputs. We will illustrate this concept by means of the sum-of-products circuit in Figure 11.5.

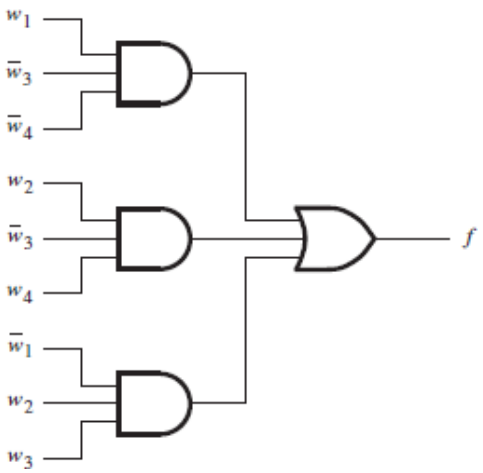


Figure 11.5 Circuit with a tree structure.

If any input of an AND gate is stuck-at-0, this condition can be detected by setting all inputs of the gate to 1 and ensuring that the other AND gates produce 0. This makes $f = 1$ in the good circuit, and $f = 0$ in the faulty circuit. Three such tests are needed because there are three AND gates.

Testing for stuck-at-1 faults is slightly more involved. An input of an AND gate is tested for the stuck-at-1 fault by driving it with the logic value 0, while the other inputs of the gate have the logic value 1. Thus a good gate produces the

output of 0, and a faulty gate generates 1. At the same time, the other AND gates must have the output of 0, which is accomplished by making at least one input of these gates equal to 0.

Figure 11.6 shows the derivation of the necessary tests. The first three tests are for the stuck-at-0 faults. Test 4 detects a stuck-at-1 fault on either the first input of the top AND gate or the third inputs of the other two gates. Observe that in each case the tested input is driven by logic 0, while the other inputs are equal to 1. This yields the test vector $w_1\ w_2\ w_3\ w_4 = 0100$. Clearly, it is useful to test inputs on as many gates as possible using a single test vector. Test 5 detects a fault on either the second input of the top gate or the first input of the bottom gate; it does not test any inputs of the middle gate. The required test pattern is 1110. Three more tests are needed to detect stuck-at-1 faults on the remaining inputs of the AND gates. Therefore, the complete test set is

$$\text{Test set} = \{1000, 0101, 0111, 0100, 1110, 1001, 1111, 0011\}$$

	No.	Product term			Test
		$w_1\overline{w}_3\overline{w}_4$	$w_2\overline{w}_3w_4$	$\overline{w}_1w_2w_3$	$w_1w_2w_3w_4$
Stuck-at-0 tests	1	1 1 1	0 1 0	0 0 0	1 0 0 0
	2	0 1 0	1 1 1	1 1 0	0 1 0 1
	3	0 0 0	1 0 1	1 1 1	0 1 1 1
Stuck-at-1 tests	4	0 1 1	1 1 0	1 1 0	0 1 0 0
	5	1 0 1	1 0 0	0 1 1	1 1 1 0
	6	1 1 0	0 1 1	0 0 0	1 0 0 1
	7	1 0 0	1 0 1	0 1 1	1 1 1 1
	8	0 0 0	0 0 1	1 0 1	0 0 1 1

Figure 11.6 Derivation of tests for the circuit in Figure 11.5.

11.5 RANDOM TESTS

So far we have considered the task of deriving a *deterministic* test set for a given circuit, primarily relying on the path-sensitizing concept. In general, it is difficult to generate such test sets when circuits become larger. A useful alternative is to choose the tests at random, which we will explore in this section.

Figure 11.7 gives all functions of two variables. For an n-variable function, there are 2^{2^n} possible functions; hence there are $2^{2^2} = 16$ two-variable functions. Consider the XOR function, implemented as shown in Figure 11.8. Let us

consider the possible stuck-at-0 and stuck-at-1 faults on wires b, c, d, h, and k in this circuit. Each fault transforms the circuit into a faulty circuit that implements a function other than XOR, as indicated in Figure 11.9. To test the circuit, we can apply one or more input valuations to distinguish the good circuit from the possible faulty circuits listed in Figure 11.9. Choose arbitrarily $w_1\ w_2 = 01$ as the first test. This test will distinguish the good circuit, which must generate $f = 1$, from the faulty circuits that realize f_0, f_2, f_3, and f_{10}, because each of these would generate $f = 0$. Next, arbitrarily choose the test $w_1\ w_2 = 11$. This test distinguishes the good circuit from the faulty circuits that realize f_5, f_7, and f_{15}, in addition to f_3, which we have already tested for using $w_1\ w_2 = 01$. Let the third test be $w_1\ w_2 = 10$; it will distinguish the good circuit from f_4 and f_{12}. These three tests, chosen in a seemingly random way, detect all faulty circuits that involve the faults in Figure 11.9. Moreover, note that the first two tests distinguish seven of the nine possible faulty circuits.

w_1w_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Figure 11.7 All two-variable functions.

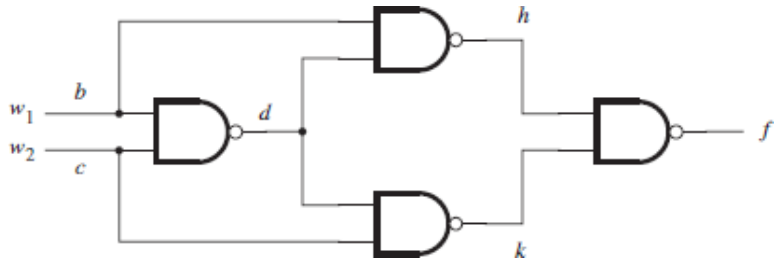


Figure 11.8 The XOR circuit.

Fault	Circuit implements
$b/0$	$f_5 = w_2$
$b/1$	$f_{10} = \overline{w}_2$
$c/0$	$f_3 = w_1$
$c/1$	$f_{12} = \overline{w}_1$
$d/0$	$f_0 = 0$
$d/1$	$f_7 = w_1 + w_2$
$h/0$	$f_{15} = 1$
$h/1$	$f_4 = \overline{w}_1 w_2$
$k/0$	$f_{15} = 1$
$k/1$	$f_2 = w_1 \overline{w}_2$

Figure 11.9 The effect of various faults.

This example suggests that it may be possible to derive a suitable test set by selecting the tests randomly. How effective can random testing be? Looking at Figure 11.7, we see that any of the four possible tests distinguishes the correct function from eight faulty functions, because they produce different output values for this input valuation. These eight faulty functions detectable by a single test are one-half of the total number of possible functions (2^{2^2-1} for the two-variable case). The test cannot distinguish between the correct function and the seven faulty functions that produce the same output value. The application of the second test distinguishes four of the remaining seven functions because they produce an output value that is different from the correct function. Thus each application of a new test essentially cuts in half the number of faulty functions that have not been detected. Consequently, the probability that the first few tests will detect a large portion of all possible faults is high. More specifically, the probability that each faulty circuit can be detected by the first test is

$$P_1 = \frac{1}{2^{2^2} - 1} \cdot 2^{2^2-1} = \frac{8}{15} = 0.53$$

This is the ratio of the number of faulty circuits that produce an output value different from the good circuit, to the total number of faulty circuits.

This reasoning is readily extended to *n*-variable functions. In this case the first test detects 2^{2^n-1} out of a total of $2^{2^n} - 1$ possible faulty functions. Therefore, if *m* tests are applied, the probability that a faulty circuit will be detected is

$$P_m = \frac{1}{2^{2^n} - 1} \cdot \sum_{i=1}^{m} 2^{2^n-i}$$

This expression is depicted in graphical form in Figure 11.10. The conclusion is that random testing is very effective and that after a few tens of tests the existence of a fault is likely to be detected even in very large circuits.

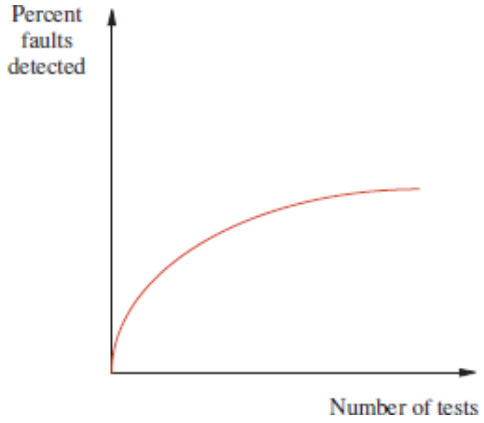


Figure 11.10 Effectiveness of random testing.

Random testing works particularly well for circuits that do not have high fan-in. If fan-in is high, then it may be necessary to resort to other testing schemes. For example, suppose that an AND gate has a large number of inputs. Then there is a problem with detecting stuck-at-1 faults on its inputs, which may not be covered by random tests. But it is possible to test for these faults using the approach described in Section 11.4.

The simplicity of random testing is a very attractive feature. For this reason, coupled with good effectiveness of tests, this technique is often used in practice.

11.6 Testing of Sequential Circuits

As seen in the previous sections, combinational circuits can be tested effectively, using either deterministic or random test sets. It is much more difficult to test sequential circuits. The presence of memory elements allows a sequential circuit to be in various states, and the response of the circuit to externally applied test inputs depends on the state of the circuit.

A combinational circuit can be tested by comparing its behavior with the functionality specified in the truth table. An equivalent attempt would be to test a sequential circuit by comparing its behavior with the functionality specified in the state table. This entails checking that the circuit performs correctly all transitions between states and that it produces a correct output. This approach may seem easy, but in reality it is extremely difficult. A big problem is that it is difficult to

ascertain that the circuit is in a specific state if the state variables are not observable on the external pins of the circuit, which is usually the case. Yet for each transition to be tested, it is necessary to verify with complete certainty that the correct destination state was reached. Such an approach may work for very small sequential circuits, but it is not feasible for practical-size circuits. A much better approach is to design the sequential circuits so that they are easily testable.

11.6.1 DESIGN FOR TESTABILITY

A synchronous sequential circuit comprises the combinational circuit that implements the output and next-state functions, as well as the flip-flops that hold the state information during a clock cycle. A general model for the sequential circuits is shown in Figure 6.87. The inputs to the combinational network are the primary inputs, w_1 through w_n, and the present state variables, y_1 through y_k. The outputs of the network are the primary outputs, z_1 through z_m, and the next-state variables, Y_1 through Y_k. The combinational network could be tested using the techniques presented in the previous sections if it were possible to apply tests on all of its inputs and observe the results on all of its outputs. Applying the test vectors to the primary inputs poses no difficulty. Also, it is easy to observe the values on the primary outputs. The question is how to apply the test vectors on the present-state inputs and how to observe the values on the next-state outputs.

A possible approach is to include a two-way multiplexer in the path of each present-state variable so that the input to the combinational network can be either the value of the state variable (obtained from the output of the corresponding flip-flop) or the value that is a part of the test vector. A significant drawback of this approach is that the second input of each multiplexer must be directly accessible through external pins, which requires many pins if there are many state variables. An attractive alternative is to provide a connection that allows shifting the test vector into the circuit one bit at a time, thus trading off pin requirements for the time it takes to perform a test. Several such schemes have been proposed, one of which is described below.

Scan-Path Technique

A popular technique, called the *scan path*, uses multiplexers on flip-flop inputs to allow the flip-flops to be used either independently during normal operation of the sequential circuit, or as a part of a shift register for testing purposes. Figure 11.11 presents the general scan-path structure for a circuit with three flip-flops. A 2-to-1 multiplexer connects the D input of each flip-flop either to the corresponding next-state variable or to the serial path that connects all flip-flops

into a shift register. The control signal $\overline{Normal}/Scan$ selects the active input of the multiplexer. During the normal operation the flip-flop inputs are driven by the next-state variables, Y_1, Y_2, and Y_3.

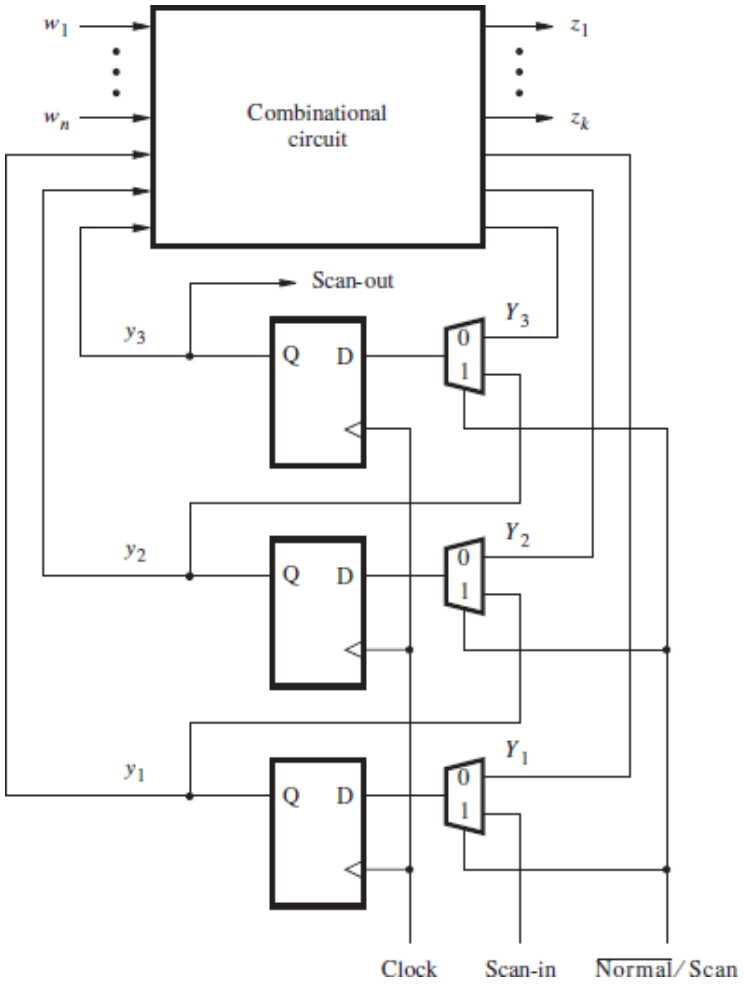


Figure 11.11 Scan-path arrangement.

For testing purposes the shift-register connection is used to scan in the portion of each test vector that involves the present-state variables, y_1, y_2, and y_3. This connection has Q_i connected to D_{i+1}. The input to the first flip-flop is the externally accessible pin *Scan-in*. The output comes from the last flip-flop, which is provided on the *Scan-out* pin.

The scan-path technique involves the following steps:

1. The operation of the flip-flops is tested by scanning into them a pattern of 0s and 1s, for example, 01011001, in consecutive clock cycles, and observing whether the same pattern is scanned out.

2. The combinational circuit is tested by applying test vectors on $w_1 w_2 \cdots w_n y_1 y_2 y_3$ and observing the values generated on $z_1 z_2 \cdots z_m Y_1 Y_2 Y_3$. This is done as follows:
   - The $y_1 y_2 y_3$ portion of the test vector is scanned into the flip-flops during three clock cycles, using $\overline{Normal}/Scan = 1$.
   - The $w_1 w_2 \cdots w_n$ portion of the test vector is applied as usual and the normal operation of the sequential circuit is performed for one clock cycle, by setting $\overline{Normal}/Scan = 0$. The outputs $z_1\ z_2 \cdots z_m$ are observed. The generated values of $Y_1\ Y_2\ Y_3$ are loaded into the flip-flops at this time.
   - The select input is changed to $\overline{Normal}/Scan = 1$, and the contents of the flip-flops are scanned out during the next three clock cycles, which makes the $Y_1\ Y_2\ Y_3$ portion of the test result observable externally. At the same time, the next test vector can be scanned in to reduce the total time needed to test the circuit.

The next example shows a specific circuit that is designed for scan-path testing.

Example 11.3

Figure 6.77 shows a circuit that recognizes a specific input sequence, which was discussed in Section 6.9. The circuit can be made easily testable by modifying it for scan path as shown in Figure 11.12. The combinational part, consisting of four AND and two OR gates, is the same in both figures.

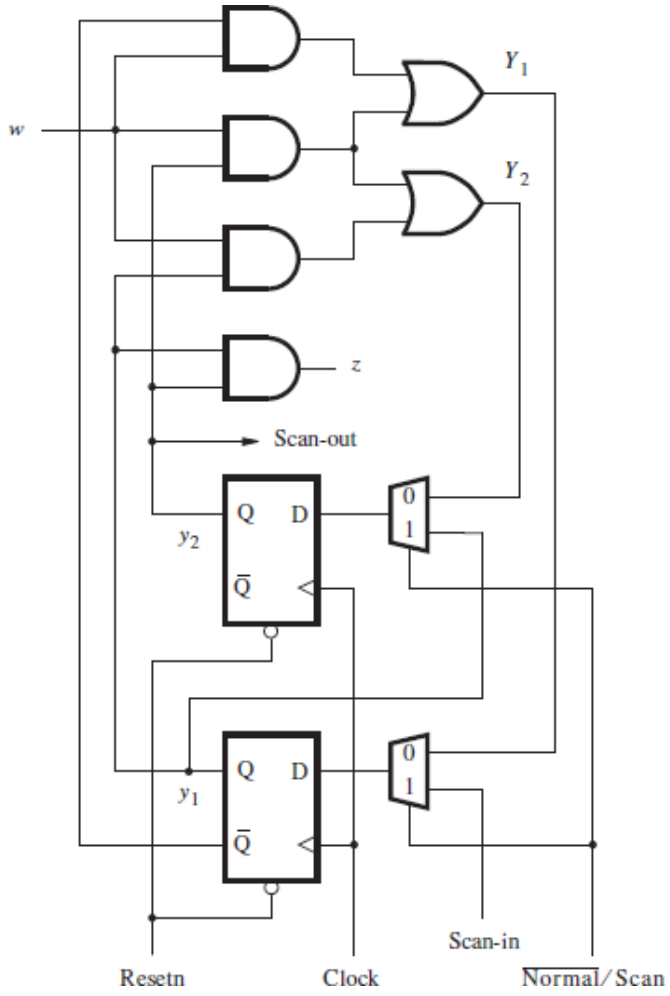


Figure 11.12 Circuit for Example 11.3.

The flip-flops can be tested by scanning through them a sequence of 0s and 1s as explained above. The combinational circuit can be tested by applying test vectors on w, y_1, and y_2. Let us use the random-testing approach, choosing arbitrarily four test vectors $w\, y_1\, y_2$ = 001, 110, 100, and 111. To apply the first test vector, the pattern $y_1\ y_2$ = 01 is scanned into the flip-flops during two clock cycles. Then for one clock cycle, the circuit is made to operate in the normal mode with w = 0. This essentially applies the vector $w\, y_1\, y_2$ = 001 to the AND-OR circuit. The result of this test should be z = 0, Y_1 = 0, and Y_2 = 0. The value of z can be observed directly. The values of Y_1 and Y_2 are loaded into the respective flip-flops, and they are scanned out during the next two clock cycles. As these values are being scanned out, the next test pattern $y_1\ y_2$ = 10 can be scanned in. Thus it takes five cycles to perform one test, but the last two cycles are

overlapped with the second test. The third and fourth tests are performed in the same way. The total time needed to perform all four tests is 14 clock cycles.

The preceding approach is based on testing a sequential circuit by testing its combinational part using the techniques developed in the previous sections. The scan-path facility makes it also possible to test the sequential circuit by making it go through all transitions specified in the state table. The circuit can be placed into a given state simply by scanning into the flip-flops the valuation of the state variables that denotes this state. The result of the transition can be checked by observing the primary outputs and by scanning out the valuation that presents the destination state. We leave it to the reader to develop the details of this approach (see Problem 11.16).

One limitation of the scan-path technique is that it does not work well if the asynchronous preset and reset features of the flip-flops are used during normal operation. We have already suggested that it is better to use synchronous preset and reset. If the designer wishes to use the asynchronous preset and reset capability, then a testable circuit can be designed using techniques such as the *level-sensitive scan design* [2, 3]. The reader can consult the references for a description of this technique

11.7 Built-in Self-Test

Until now we have assumed that testing of logic circuits is done by externally applying the test inputs and comparing the results with the expected behavior of the circuit. This requires connecting external equipment to the circuit under test. An interesting question is whether it is possible to incorporate the testing capability within the circuit itself so that no external equipment is needed. Such built-in capability would allow the circuit to be self-testable. This section presents a scheme that provides the *built-in self-test* (BIST) capability.

Figure 11.13 shows a possible BIST arrangement in which a test vector generator produces the test vectors that must be applied to the circuit under test. In Section 11.5 we explained that randomly chosen test vectors give good results, with the fault coverage depending on the number of tests performed. For each test vector applied to the circuit, it is necessary to determine the required response of the circuit. The response of a good circuit may be determined using the simulator tool of a CAD system. The expected responses to the applied tests must be stored on the chip so that a comparison can be made when the circuit is being tested.

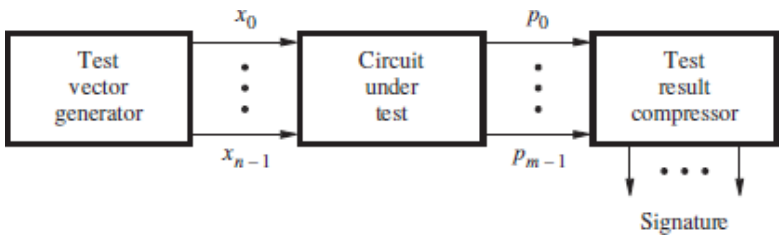
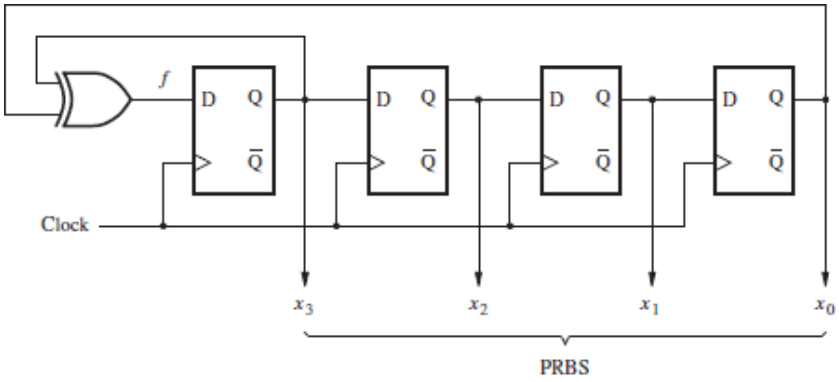


Figure 11.13 The testing arrangement.

A practical approach for generating the test vectors on-chip is to use *pseudorandom* tests, which have the same characteristics as random tests but are produced deterministically and can be repeated at will. The generator for pseudorandom tests is easily constructed using a feedback shift-register circuit. A small example of a possible generator is given in Figure 11.14. A four-bit shift register, with the signals from the first and fourth stages fed back through an XOR gate, generates 15 different patterns during successive clock cycles. If the shift register is set at the beginning to $x_3x_2x_1x_0 = 1000$, then the generated patterns are as shown in part (*b*) of the figure. Observe that the pattern 0000 cannot be used, because the circuit would be locked in this pattern indefinitely.



(a) Circuit

x_3	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0	1	...
x_2	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0	...
x_1	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	...
x_0	0	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0	...
f	1	1	1	0	1	0	1	1	0	0	1	0	0	0	1	1	...

(b) Generated sequence

Figure 11.14 Pseudorandom binary sequence generator (PRBSG).

The circuit in Figure 11.14 is representative of a class of circuits known as *linear feedback shift registers* (LFSRs). Using feedback from the various stages of an *n*-bit shift register, connected to the first stage by means of XOR gates, it is possible to generate a sequence of $2^n - 1$ patterns that have the characteristics of randomly generated numbers. Such circuits are used extensively in error-correcting codes.

The pseudorandom binary sequence generator (PRBSG) gives a simple method of generating tests. The required response of the circuit under test can be determined by using the simulator tool of the CAD system. The remaining question is how to check whether the circuit indeed produces the required response. It is not attractive to have to store a large number of responses to the tests on a chip that also includes the main circuit. A practical solution is to compress the results of the tests into a single pattern. This can be done using an LFSR circuit. Instead of just providing the feedback signals as the input, a compressor circuit includes the output signals produced by the circuit under test. Figure 11.15 shows a single-input compressor circuit (SIC), which uses the same feedback connections as the PRBSG of Figure 11.14. The input *p* is the output of a circuit under test. After applying a number of test vectors, the resulting values of *p* drive the SIC and, coupled with the LFSR functionality, produce a four-bit pattern. The pattern generated by the SIC is called a *signature* of the tested circuit for the given sequence of tests. The signature represents a single pattern that may be interpreted as a result of all the applied tests. It can be compared against a predetermined pattern to see if the tested circuit is working properly. Storing a single *n*-bit pattern for comparison purposes presents only a small overhead. The randomizing nature of the compressor circuits based on LFSRs provides a good coverage of patterns that may result from a faulty circuit.

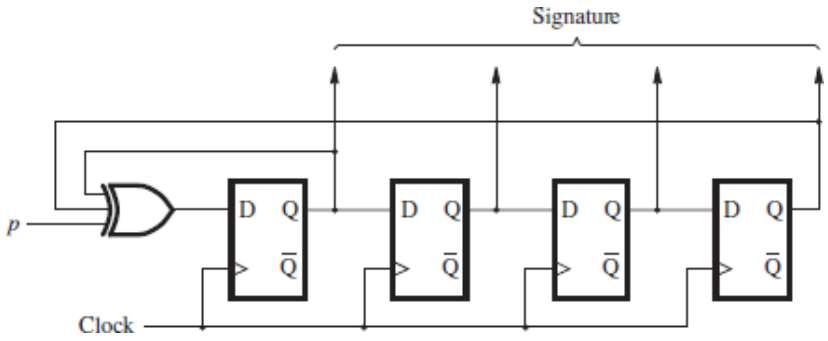


Figure 11.15 Single-input compressor circuit (SIC).

If the circuit under test has more than one output, then an LSFR with multiple inputs can be used. Figure 11.16 illustrates how four inputs, p_0 through p_3, can be added to the basic circuit of Figure 11.14. Again the four-bit signature provides a

good mechanism for distinguishing among different sequences of four-bit patterns that may appear on the inputs of this multiple-input compressor circuit (MIC).

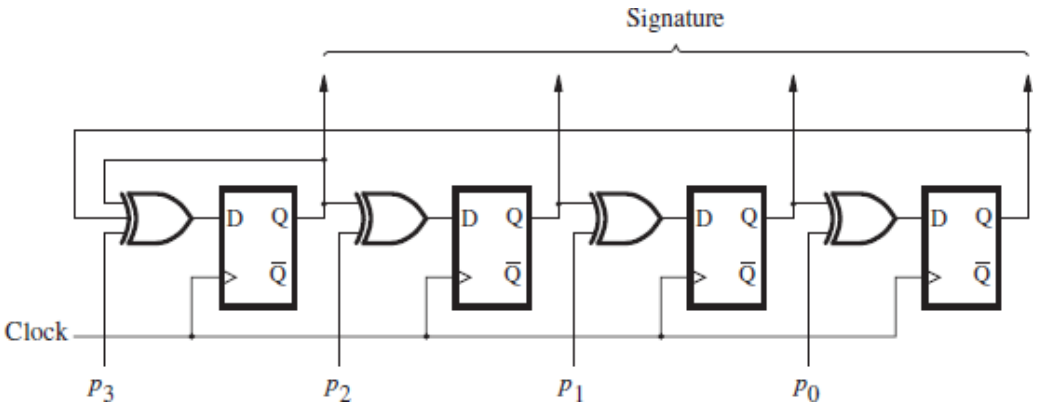


Figure 11.16 Multiple-input compressor circuit (MIC).

A complete BIST scheme for a sequential circuit may be implemented as indicated in Figure 11.17. The scan-path approach is used to provide a testable circuit. The test patterns that would normally be applied on the primary inputs $W = w_1\ w_2\ \cdots\ w_n$ are generated internally as the patterns on $X = x_1\ x_2\ \cdots\ x_n$. Multiplexers are needed to allow switching from W to X, as inputs to the combinational circuit. A pseudorandom binary sequence generator, PRBSG-X, generates the test patterns for X. The portion of the tests applied via the next-state signals, y, is generated by the second PRBS generator, PRBSG-y. These patterns are scanned into the flip-flops as explained in Section 11.6.

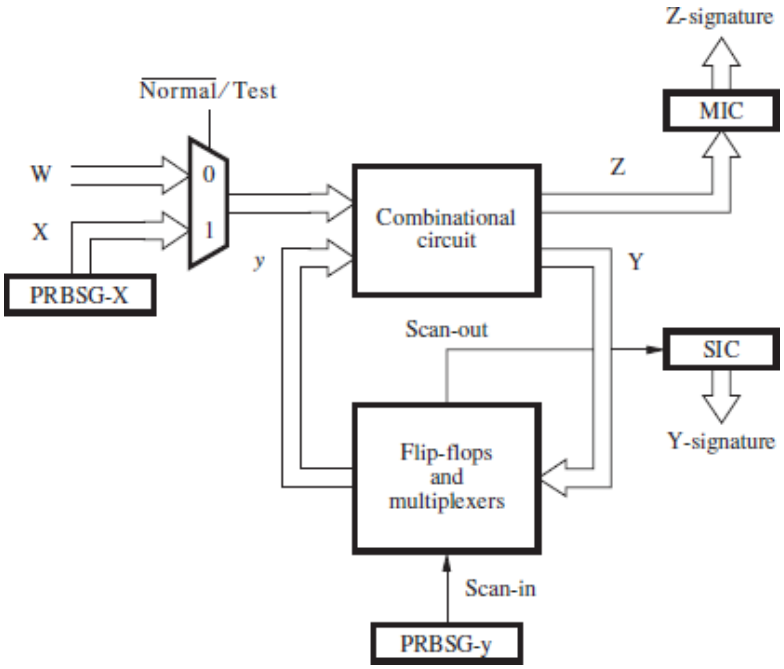


Figure 11.17 BIST in a sequential circuit.

The test outputs are compressed using the two compressor circuits. The patterns on the primary outputs, $Z = z_1\ z_2 \cdots z_m$, are compressed using the MIC circuit, and those on the next-state wires $Y = Y_1\ Y_2 \cdots Y_k$, by the SIC circuit. These circuits produce the *Z*-signature and *Y*-signature, respectively. The testing procedure is the same as given in Example 11.3 except that the comparison with the test result that a good circuit is supposed to give is done only once; at the end of the testing process the two signatures are compared with the stored patterns. Figure 11.17 does not show the circuitry needed to store these patterns and perform the comparison. Instead of storing the signature patterns of the required results as a part of the designed circuit, it is possible to shift out the contents of MIC and SIC shift registers onto two output pins and to perform the necessary comparison with the expected signatures externally. Note that using signature testing in this way reduces the testing time significantly, compared to the time it would take to test the circuit by scanning out the results of individual tests and comparing them with predetermined patterns.

The effectiveness of the BIST approach depends on the length of the LFSR generator and compressor circuits. Longer shift registers give better results. One reason for failing to detect that the circuit under test may be faulty is that the pseudorandomly generated tests do not have perfect coverage of all possible faults. Another reason is that a signature generated by compressing the outputs of a faulty circuit may coincidentally end up being the same as the signature of the good circuit. This can occur because the compression process results in a loss of some information, such that two distinct output patterns may be compressed into the same signature. This is known as the *aliasing* problem.

11.7.1 Built-in Logic Block Observer

The essence of BIST is to have internal capability for generation of tests and for compression of the results. Instead of using separate circuits for these two functions, it is possible to design a single circuit that serves both purposes. Figure 11.18 shows the structure of a possible circuit, known as the *built-in logic block observer* (BILBO) [4]. This four-bit circuit has the same feedback connections as the circuit of Figure 11.14.

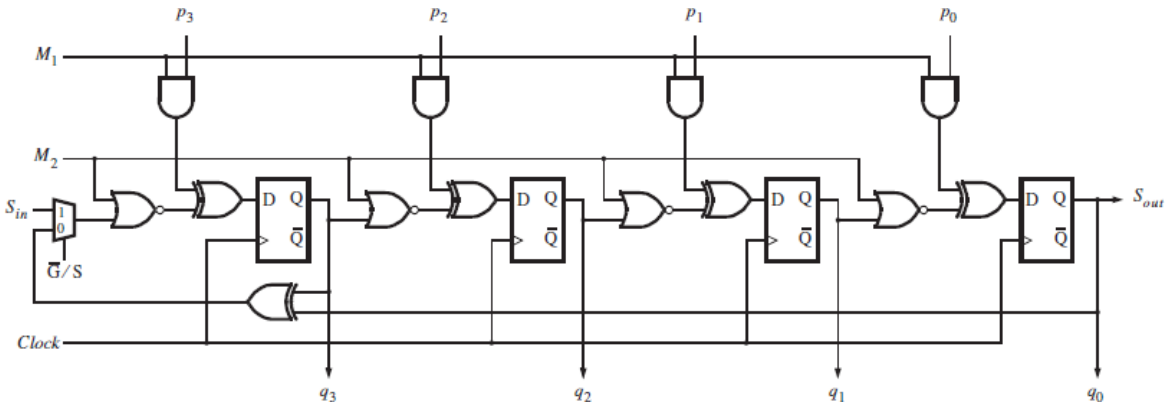


Figure 11.18 A four-bit built-in logic block observer (BILBO).

The BILBO circuit has four modes of operation, which are controlled by the mode bits, M_1 and M_2. The modes are as follows:

- $M_1\ M_2 = 11$ — Normal system mode in which all flip-flops are independently controlled by the signals on inputs p_0 through p_3. In this mode each flip-flop may be used to implement a state variable of a finite state machine by using p_0 to p_3 as y_0 to y_3.
- $M_1\ M_2 = 00$ — Shift-register mode in which the flip-flops are connected into a shift register. This mode allows test vectors to be scanned in, and the results of applied tests to be scanned out, if the control input $\overline{G}/S$ is equal to 1. If $\overline{G}/S = 0$, then the circuit acts as the PRBS generator.
- $M_1\ M_2 = 10$ — Signature mode in which a series of patterns applied on inputs p_0 through p_3 are compressed into a signature available as a pattern on q_0 through q_3.
- $M_1\ M_2 = 01$ — Reset mode in which all flip-flops are reset to 0.

An efficient way of using BILBO circuits is presented in Figure 11.19. A combinational circuit can be tested by partitioning it into two (or more) parts. A BILBO circuit is used to provide inputs to one part and to accept outputs from the other part. The testing process involves a two-phase approach. First, BILBO1 is used as a PRBS generator that provides test patterns for combinational network 1 (CN1). During this time BILBO2 acts as a compressor and produces a signature for the test. The signature is shifted out by placing BILBO2 into the shift-register mode. Next, the roles of BILBO1 and BILBO2 are reversed, and the process is repeated to test CN2.

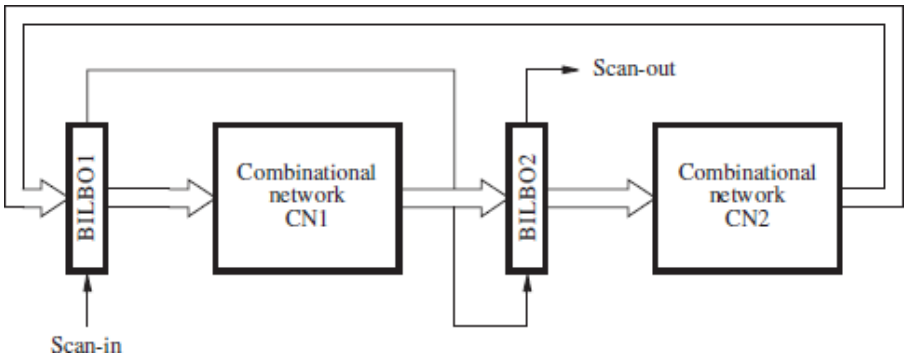


Figure 11.19 Using BILBO circuits for testing.

The detailed steps in the testing process are

1. Scan the initial test pattern into BILBO1 and reset all flip-flops in BILBO2.
2. Use BILBO1 as the PRBS generator for a given number of clock cycles and use BILBO2 to produce a signature.
3. Scan out the contents of BILBO2 and externally compare the signature; then scan into it the initial test pattern for testing CN2. Reset the flip-flops in BILBO1.
4. Use BILBO2 as the PRBS generator for a given number of clock cycles and use BILBO1 to produce a signature.
5. Scan out the signature in BILBO1 and externally compare it with the required pattern.

The BILBO circuits are used in this way for testing purposes. At other times the normal system mode is used.

11.7.2 SIGNATURE ANALYSIS

We have explained the use of signatures in the context of implementing an efficient built-in testing mechanism. The main idea of compressing a long sequence of test results into a single signature was originally developed as the basis for an instrument manufactured by Hewlett-Packard in the 1970s, known as the Signature Analyzer [5]. Thus the name *signature analysis* was coined to refer to the testing schemes that use signatures to represent the results of applied tests.

Signature analysis is particularly suitable for digital systems that naturally include an ability to generate the desired test patterns. Such is the case with computer systems in which various parts of the system can be stimulated by test patterns produced under software control.

11.7.3 Boundary Scan

The testing techniques discussed in the previous sections are equally applicable to circuits that are implemented on single chips or on printed circuit boards that contain a number of chips. A circuit can be tested only if it is possible to apply the tests to it and observe the outputs produced. This involves having access to the primary inputs and outputs.

When chips are soldered onto a printed circuit board, it often becomes impossible to attach test probes to pins. This hinders the testing process unless some indirect access to the pins is provided. The scan-path concept can be extended to the board level to deal with the problem. Suppose that each primary input or output pin on a chip is connected through a D flip-flop and that a provision is made for a test mode in which all flip-flops can be connected into a shift register. Then the test information can be scanned in and scanned out using the shift-register path, via two pins that serve as serial input and output. Connecting the serial output pin of one chip to the serial input pin of another chip results in the pins of all chips being connected into a board-wide shift register for testing purposes. This approach has become popular in practice and has been embodied into the IEEE Standard 1149.1 [6].

11.8 Printed Circuit Boards

Design and testing techniques presented in this book can be applied to any logic circuit, whether the circuit is realized on a single chip or its implementation involves a number of chips placed on a printed circuit board (PCB). In this section we discuss some practical issues that arise when one or more circuits that form a larger digital system are implemented on a PCB.

A typical PCB contains multiple layers of wiring. When the board is manufactured, the wiring pattern on each layer is generated. The layers are separated by insulating material and pressed together in sandwichlike fashion to form the board. Connections between different wiring levels are made through holes that are provided for this purpose. Chips and other components are then soldered to the top and possibly to the bottom layers.

In preceding chapters we have discussed in considerable detail the CAD tools used for designing circuits that can be implemented on a single chip, such as an FPGA. For a multiple-chip implementation, we need a different set of CAD tools to design a PCB that incorporates the chips and connections needed to realize the complete digital system. Such tools are available from a number of companies,

for example, Cadence Design Systems and Mentor Graphics. These tools can automatically determine where each chip should be placed on the PCB, but the designer can also specify the location of particular chips. This is called the *placement* process. Given a specific placement of chips and other components (such as connectors and capacitors), the tools generate a layout for each layer of wiring traces that provide the required connections on the board. This process is referred to as *routing*. Again the designer can intervene and manually route some connections. However, since the number of connections can be in the tens of thousands, it is crucial to obtain a good automated solution.

In addition to the design issues discussed in the previous chapters, a large circuit implemented on a PCB is subject to some other constraints. Signals on the wiring traces may be affected by noise problems caused by crosstalk, spikes in the power supply voltage, and reflections from the end points of long traces.

Crosstalk

Two closely spaced wires that run parallel to each other are capacitively coupled, and a pulse on one wire can induce a similar (but usually much smaller) pulse on the adjoining wire. This is referred to as *crosstalk*. Its existence is undesirable because it contributes to noise problems.

When drawing timing diagrams, we usually draw ideal waveforms with sharp edges, which have well-defined voltage levels for the logic values 0 and 1. In an actual circuit the corresponding signals may depart significantly from the desired behavior. As explained in Appendix B, noise in a circuit can affect voltage levels, which can be troublesome. For example, if at some point in time the noise diminishes the value of a signal that should be at logic 1 to a level where this signal is interpreted by the next gate as being logic 0, then a malfunction in the circuit is likely to occur. Since the noise effects tend to be random, they are often difficult to detect.

To minimize crosstalk, it is prudent to avoid having long wires running parallel in close proximity to each other. This may be difficult to achieve because of limited space on a PCB and the need to provide a large number of wires. Using additional layers (planes) of wiring helps in coping with crosstalk problems.

Power Supply Noise

When a CMOS circuit changes its state, there is a momentary flow of current in the circuit, which is manifested as a current pulse on the power supply (V_{DD} and *Ground*) wires. Since a wiring trace on a PCB has a small “line inductance,” such a current pulse causes a voltage spike (short pulse) on these lines. The cumulative effect of such voltage spikes can cause a malfunction of the circuit.

The induced voltage spikes can be reduced significantly by connecting a small capacitor between the V_{DD} and *Ground* wires, in close proximity to the chip that causes the spikes to occur. Since these spikes have the characteristic of a very high frequency signal, the path through the capacitor is essentially a short circuit for them. Thus the voltage spikes "bypass" the power supply lines and do not affect other chips connected to the same lines. Such capacitors are called *bypass capacitors*. They do not affect the DC voltage on the power supply lines.

Reflections and Terminations

Wiring traces on a PCB act as simple wires in circuits when the clock frequency is low. However, at higher clock frequencies it becomes necessary to worry about so-called *transmission-line effects*. When a signal propagates along a long wire, it is attenuated due to the small resistance of the wire, it picks up crosstalk that manifests itself as noise, and it may be reflected when it reaches the end of the wire. The reflection causes a problem if its effect does not die down before the next active clock edge. The discussion of transmission-line effects is beyond the scope of this book. We will only mention that the reflection of signals can be prevented by placing a suitable "termination" component on the line. This termination can be as simple as a resistor whose resistance matches the apparent resistance of the line, known as the *characteristic impedance* of the line. Other forms of termination are also possible.

11.8.1 TESTING OF PCBS

The manufactured PCB has to be tested thoroughly. Flaws in the manufacturing process may cause some connections to be broken and others to be shorted by a solder blob that touches two adjacent wires. There may be problems caused by design errors that were not discovered during the design process. Finally, some chips and other components on the PCB may be defective.

Power Up

The first step is to turn on the power supply. In the worst case this may cause some chip to be destroyed because of a fatal short-circuit condition (in an extreme case a chip package may actually blow apart). Assuming that this is not the case, it is essential to check if any of the chips is becoming inordinately hot. Overheating is a symptom of a serious problem that must be corrected.

It is also necessary to check that the power and ground connections are properly made on each chip and that the voltage level is as specified.

Reset

The next step is to reset all circuitry on the PCB to reach a predetermined starting point. This typically implies resetting the flip-flops, which is usually achieved by asserting a common reset line. It is important to verify that the starting state is correctly established.

Low-Level Functional Testing

Since practical circuits can be extremely complex, it is prudent to test the basic functionality first. A key test is to verify that the control signals are working correctly.

Using the divide-and-conquer approach, simple functions are tested first, followed by the more complex ones.

Full Functional Testing

Having verified the operation of smaller subcircuits, it is necessary to test the functionality of the entire system on the PCB. The number of errors often depends on the thoroughness of the simulation done during the design process. In general, it is difficult to simulate large digital systems fully, so some errors are likely to be found on the PCB. Typical errors are due to

- Manufacturing errors, such as wrong wiring traces, blown components, or incorrect power supply voltage.
- Incorrect specifications.
- Designer's misinterpretation of information on the data sheets that describe some chips.
- Incorrect information on the data sheets provided by the chip manufacturer.

As mentioned earlier, PCBs contain multiple layers of wiring. Each layer may have several thousands of wires in it. Finding and fixing errors can be a difficult and time-consuming task, especially if errors involve wires in internal (as opposed to the top or bottom) wiring layers.

Timing

It is next necessary to verify the timing of the circuit. A good strategy is to start with a slow clock. If the circuit works properly, then the clock frequency is gradually increased until the required operating frequency is reached.

Timing problems arise because of propagation delays through various paths in a circuit. These delays are caused by the logic gates and the wiring that

interconnects them. It is essential to ensure that all data inputs to flip-flops in the circuit are stable before the active edge of the clock signal arrives, as required by the setup time.

Reliability

A digital system is expected to operate reliably for a long time. Its reliability may be affected by several factors, such as timing, noise, and crosstalk problems.

The timing of signals has to provide some safety margin to allow for small changes in propagation delays. If the timing is too tight, then it is likely that the circuit will operate correctly for some period of time, but will eventually fail because of a timing error. The timing of chips may change with temperature, so failures can occur if thermal constraints are not adhered to. Cooling is usually provided by means of fans.

11.8.2 INSTRUMENTATION

Testing of circuits implemented in PCBs requires some specialized instruments.

Oscilloscope

The details of individual signals can be examined using an oscilloscope. This instrument displays the voltage waveform of a signal, showing the potential problems with respect to propagation delay and noise. The waveform displayed on an oscilloscope shows the actual voltage levels of the signal; it does not depict the simplified view of ideal waveforms that have perfectly square edges. If the user wants to see only the logic values of a signal (0 or 1), then a different type of instrument called a logic analyzer can be used.

Logic Analyzer

While an oscilloscope allows simultaneous examination of a few signals, a logic analyzer allows examination of tens or even hundreds of signals at the same time. It takes inputs from a set of points in the circuit, by means of probes attached to these points, and digitizes and displays the detected signals in the form of waveforms on a screen. A powerful feature of the logic analyzer is that it has internal storage capable of recording a sequence of changes in the signals over a substantial period of time. Then any segment of this information can be displayed as desired by the operator. Typically, it is possible to record a few milliseconds' worth of events, which involves many cycles of a normal digital clock.

Looking at the waveforms taken when the circuit under test is working properly is not helpful in the debugging process. It is essential to see the waveforms generated when a malfunction takes place. The logic analyzer can be "triggered" to record a window of events that occurred before and after the trigger event. The user must specify the trigger event. For example, suppose that a malfunction is suspected to be caused by two control signals, *A* and *B*, being asserted at the same time, even though the design specification requires that these signals be mutually exclusive. A useful trigger point can then be established as the time when the AND of *A* and *B* has the value 1. Finding suitable trigger events can be difficult, and the user must rely on intuition and experience.

To use a logic analyzer effectively, it must be possible to connect the probes to some useful (for testing purposes) points in the circuit. Thus it is important to provide such "test" points when a PCB is being designed.

11.9 Concluding Remarks

Manufactured products must be tested to ensure that they perform as expected. All of the techniques discussed in this chapter are relevant for this type of testing. The development of tests and the required responses is based on the assumption that the circuits are designed correctly. Thus it is the validity of the physical implementation that is being tested.

Another aspect of testing occurs during the design process. The designer has to ascertain that the designed circuit meets the specifications. From the testing point of view, this poses a significant problem because there exists no provably good circuit that can be used to generate the desired tests. CAD tools are helpful in deriving tests for a designed circuit, but they cannot determine whether the circuit is indeed what the designer intended to achieve in terms of its functionality. A design error usually results in a circuit that has somewhat different functionality than required by the specification.

Small circuits can be tested fully to verify their functionality. A combinational circuit can be tested to see if it performs according to its truth table. A sequential circuit can be tested by checking the transitions specified in the state table. This is much easier to do if the circuit is designed for testability, as explained in Section 11.6.1. Large circuits cannot be tested exhaustively, because a vast number of tests would have to be applied. In this case the designer's ingenuity is needed to determine a manageable set of tests that will hopefully demonstrate the correctness of the circuit.

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

***11.1** Derive a table similar to Figure 11.1*b* for the circuit in Figure P11.1 to show the coverage of the various stuck-at-0 and stuck-at-1 faults by the eight possible tests. Find a minimal test set for this circuit.

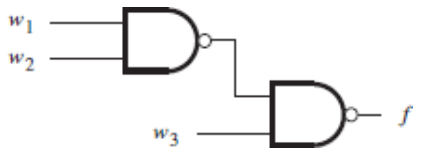


Figure P11.1 Circuit for Problem 11.1.

11.2 Repeat Problem 11.1 for the circuit in Figure P11.2.

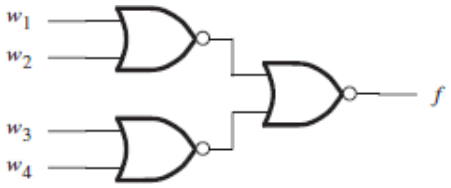


Figure P11.2 Circuit for Problem 11.2.

***11.3** Devise a test to distinguish between two circuits that implement the following expressions

$$f = x_1x_2x_3 + x_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_4 + \bar{x}_1x_3\bar{x}_4$$
$$g = (\bar{x}_1 + x_2)(x_3 + x_4)$$

11.4 Consider the circuit in Figure P11.3. Sensitize each path in this circuit to obtain a complete test set that comprises a minimum number of tests.

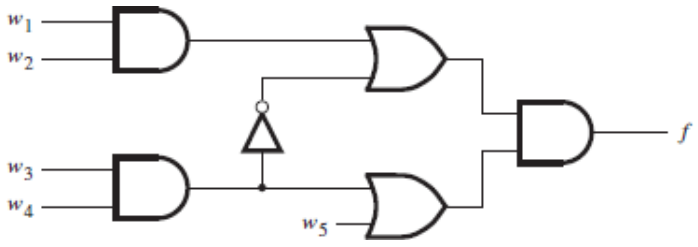


Figure P11.3 Circuit for Problem 11.4.

***11.5** For the circuit of Figure 11.4*a*, show the tests that can detect each of the faults: $w_1/0$, $w_4/1$, $g/0$, and $c/1$.

11.6 Suppose that the tests $w_1\ w_2\ w_3\ w_4 = 0100$, 1010, 0011, 1111, and 0110 are chosen randomly to test the circuit in Figure 11.3. What percentage of single faults are detected using these tests?

11.7 Repeat Problem 11.6 for the circuit in Figure 11.4*a*.

11.8 Repeat Problem 11.6 for the circuit in Figure 11.5.

***11.9** Consider the circuit in Figure P11.4. Are all single stuck-at-0 and stuck-at-1 faults in this circuit detectable? If not, explain why.

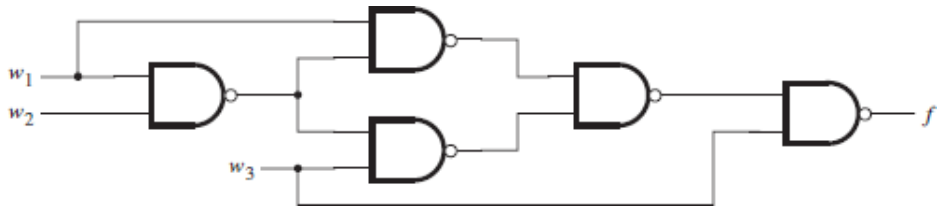


Figure P11.4 Circuit for Problem 11.9.

11.10 Prove that in a circuit in which all gates have a fan-out of 1, any set of tests that detects all single faults on the input wires detects all single faults in the entire circuit.

***11.11** The circuit in Figure P11.5 determines the parity of a four-bit data unit. Derive a minimal test set that can detect all single stuck-at-0 and stuck-at-1 faults in this circuit. Would your test set work if the XOR gates are implemented using the circuit in Figure 8.9*c*? Can your result be extended to a general case that involves *n*-bit data units?

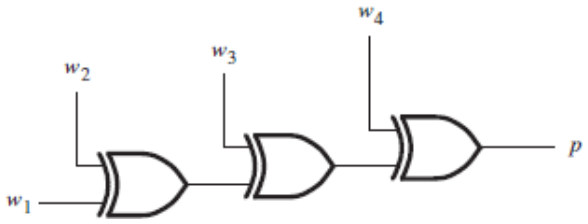


Figure P11.5 Circuit for Problem 11.11.

***11.12** Derive a test set that can detect all single faults in the decoder circuit in Figure 4.14*c*.

11.13 List all single faults in the circuit in Figure 11.4*a* that can be detected using each of the tests $w_1\ w_2\ w_3\ w_4 = 1100$, 0010, and 0110.

11.14 Sensitize each path in the combinational part of the circuit in Figure 11.12 to obtain a complete test set that comprises as few tests as possible. Show how your test set can be applied to test this circuit. How many clock cycles are needed to perform the necessary tests?

11.15 Derive an ASM chart that represents the flow of control needed to test the circuit in Figure 11.12.

11.16 The circuit in Figure 11.12 provides an easily testable implementation of the FSM in Figure 6.76. In Example 11.3 we showed how this circuit may be tested by testing the combinational part using randomly chosen tests. A different approach to testing may be to attempt to determine whether the circuit actually realizes the functionality specified in the state table in Figure 6.76*b*. This can be done by making the circuit go through all transitions given in the state table. For example, after applying the *Resetn* = 0 signal, the circuit begins in state *A*. It must be verified that the circuit is indeed forced into state *A* by scanning out the expected valuation $y_2\, y_1 = 00$. Next each transition must be checked. To verify the transition $A \rightarrow A$ if $w = 0$, it is necessary to make the input w equal to 0 and allow the normal operation to take place for one clock cycle by making $\overline{Normal}/Scan = 0$. The value of the output z must be observed. This is followed by scanning out the values of y_2 and y_1 to see if $y_2\, y_1 = 00$. At the same time, the valuation for the next test should be scanned in. If this test involves verifying that $B \rightarrow A$ if $w = 0$, then the valuation $y_2\, y_1 = 01$ is scanned in. This process continues until all transitions have been verified.

Indicate in the form of a table the values of the signals $\overline{Normal}/Scan$, *Scan-in*, *Scan-out*, w, and z, as well as the transition tested, for each clock cycle necessary to perform the complete test for this circuit.

11.17 Write VHDL code that represents the circuit in Figure 11.12.

11.18 Derive an ASM chart that describes the control needed to test a digital system that uses the BILBO structure in Figures 11.18 and 11.19.

REFERENCES

1. J. P. Roth et al., “Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits,” *IEEE Transactions on Computers* EC-16, no. 5 (October 1967), pp. 567–80.
2. A. Miczo, *Digital Logic Testing and Simulation* (Wiley: New York, 1986).
3. T. W. Williams and K. P. Parker, “Design for Testability—a Survey,” *IEEE Transactions on Computers* C-31 (January 1982), pp. 2–15.

4. B. Koenemann, J. Mucha, and G. Zwiehoff, “Built-In Logic Block Observation Techniques,” Proceedings 1977 Test Conference, IEEE Pub. 79CH1609-9C, October 1979, pp. 37–41.
5. A. Y. Chan, “Easy-to-Use Signature Analyzer Accurately Troubleshoots Complex Logic Circuits,” *Hewlett-Packard Journal*, May 1997, pp. 9–14.
6. *Test Access Port and Boundary-Scan Architecture*, IEEE Standard 1149.1, May 1990.

appendix

A

VHDL REFERENCE

This appendix describes the features of VHDL that are used in this book. It is meant to serve as a convenient reference for the reader. Hence only brief descriptions are provided, along with examples. The reader is encouraged to first study the introduction to VHDL in Sections 2.10, 3.5, and 4.6.

In some ways VHDL has an unusual syntax for describing logic circuits. The prime reason is that VHDL was originally intended to be a language for documenting and simulating circuits, rather than for describing circuits for synthesis. This appendix is not meant to be a comprehensive VHDL manual. While we discuss almost all the features of VHDL that are useful in the synthesis of logic circuits, we do not discuss any of the features that are useful only for simulation of circuits or for other purposes. Although the omitted features are not needed for any of the examples used in this book, a reader who wishes to learn more about using VHDL can refer to specialized books [1–8].

How *Not* to Write VHDL Code

In Section 2.10 we mentioned the most common problem encountered by designers who are just beginning to write VHDL code. The tendency for the novice is to write code that resembles a computer program, containing many variables and loops. It is difficult to determine what logic circuit the

CAD tools will produce when synthesizing such code. This book contains more than 150 examples of complete VHDL code that represents a wide range of logic circuits. In all of these examples, the code is easily related to the described logic circuit. The reader is encouraged to adopt the same style of code. A good general guideline is to assume that if the designer cannot readily determine what logic circuit is described by the VHDL code, then the CAD tools are not likely to synthesize the circuit that the designer is trying to describe.

Since VHDL is a complex language, errors in syntax and usage are quite common. Some problems encountered by our students, as novice designers, are listed at the end of this appendix in Section A.11. The reader may find it useful to examine these errors in an effort to avoid them when writing code.

Once complete VHDL code is written for a particular design, it is useful to analyze the resulting circuit synthesized by the CAD tools. Much can be learned about VHDL, logic circuits, and logic synthesis by studying the circuits that are produced automatically by the CAD tools.

A.1 Documentation in VHDL Code

Documentation can be included in VHDL code by writing a comment. The two characters '-' ,'-' denote the beginning of the comment. The VHDL compiler ignores any text on a line after the '--'.

Example A.1

```
-- this is a VHDL comment
```

A.2 Data Objects

Information is represented in VHDL code as data objects. Three kinds of data objects are provided: signals, constants, and variables. For describing

logic circuits, the most important data objects are signals. They represent the logic signals (wires) in the circuit. The constants and variables are also sometimes useful for describing logic circuits, but they are used infrequently.

A.2.1 DATA OBJECT NAMES

The rules for specifying data object names are simple: any alphanumeric character may be used in the name, as well as the '_' underscore character. There are four caveats. A name cannot be a VHDL keyword, it must begin with a letter, it cannot end with an '_' underscore, and it cannot have two successive '_' underscores. Thus examples of legal names are *x*, *x*1, *x_y*, and *Byte*. Some examples of illegal names are 1*x*, *_y*, *x__y*, and *entity*. The latter name is not allowed because it is a VHDL keyword. VHDL is not case sensitive. Hence x is the same as X, and ENTITY is the same as entity. To make the examples of VHDL code in this book more readable, we use uppercase letters in all keywords.

To avoid confusion when using the word signal, which can mean either a VHDL data object or a logic signal in a circuit, we sometimes write the VHDL data object as SIGNAL.

A.2.2 DATA OBJECT VALUES AND NUMBERS

We use SIGNAL data objects to represent individual logic signals in a circuit, multibit-logic signals (vectors), and integers. The value of an individual SIGNAL is specified using apostrophes, as in '0' or '1'. The value of a multibit SIGNAL is given with double quotes. An example of a four-bit SIGNAL value is "1001", and an eight-bit value is "10011000". The default number radix is binary, but a different radix can be specified explicitly, such as X"5A" (hexadecimal) or O"35" (octal). For a signal of the INTEGER type a value can be specified in decimal by not using quotes, as in 9 or 152. The values of CONSTANT or VARIABLE data objects are specified in the same way as for SIGNAL data objects.

A.2.3 SIGNAL DATA OBJECTS

SIGNAL data objects represent the logic signals, or wires, in a circuit. There are three places in which signals can be declared in VHDL code: in

an entity declaration (see Section A.4.1), in the declarative section of an architecture (see Section A.4.2), and in the declarative section of a package (see Section A.5). A signal has to be declared with an associated *type*, as follows:

```
SIGNAL signal_name : type_name ;
```

A.2.4 BIT AND BIT_VECTOR TYPES

These types are predefined in the VHDL Standards IEEE 1076 and IEEE 1164. Hence no library is needed to use these types in the code. Objects of BIT type can have the values '0' or '1'. An object of BIT_VECTOR type is a linear array of BIT objects.

Example A.2

```
SIGNAL x1   : BIT ;
SIGNAL C    : BIT_VECTOR (1 TO 4) ;
SIGNAL Byte : BIT_VECTOR (7 DOWNTO 0) ;
```

The signals *C* and *Byte* illustrate the two possible ways of defining a multibit data object. The syntax "lowest_index TO highest_index" is useful for a multibit signal that is simply an array of bits. In the signal *C* the most-significant (left-most) bit is referenced using lowest_index, and the least-significant (right-most) bit is referenced using highest_index. The syntax "highest_index DOWNTO lowest_index" is useful if the signal represents a binary number. In this case the most-significant (left-most) bit has the index highest_index, and the least-significant (right-most) bit has the index lowest_index.

The multibit signal *C* represents four BIT objects. It can be used as a single four-bit quantity, or each bit can be referred to individually. The syntax for referring to the signals individually is $C(1), C(2), C(3)$, or $C(4)$. An assignment statement such as

```
C <= "1010" ;
```

results in $C(1) = 1, C(2) = 0, C(3) = 1$, and $C(4) = 0$.

The signal *Byte* comprises eight BIT objects. The assignment statement

```
Byte <= "10011000" ;
```

results in *Byte*(7) = 1, *Byte*(6) = 0, and so on to *Byte*(0) = 0.

A.205 STD_LOGIC AND STD_LOGIC_VECTOR TYPES

The STD_LOGIC type was added to the VHDL Standard in IEEE 1164. It provides more flexibility than the BIT type. To use this type, we must include the two statements

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
```

These statements provide access to the *std_logic_1164* package, which defines the STD_ LOGIC type. We describe VHDL packages in Section A.5. In general, they are used as a place to store VHDL code, such as the code that defines a type, which can then be used in other source-code files. The following values are legal for a STD_LOGIC data object: 0, 1, Z, $-$, L, H, U, X, and W. Only the first four are useful for synthesis of logic circuits. The value Z represents high impedance, and $-$ stands for “don't care.” The value L means “weak 0,” H is “weak 1,” U is “uninitialized,” X is “unknown,” and W is “weak unknown.” The STD_LOGIC_VECTOR type represents an array of STD_LOGIC objects.

Example A.3

```
SIGNAL x1, x2, Cin, Cout, Sel  : STD_LOGIC ;
SIGNAL C                       : STD_LOGIC_VECTOR (1 TO 4) ;
SIGNAL X, Y, S                 : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
```

STD_LOGIC objects are often used in logic expressions in VHDL code. A related package is

```
USE ieee.std_logic_signed.all ;
```

The *std_logic_signed* package specifies that it is legal to use the STD_LOGIC_VECTOR signals with arithmetic operators, like + (see Section A.7.1). The VHDL compiler should generate a circuit that works

for signed numbers. An alternative is to use the package *std_logic_unsigned*. In this case the compiler should generate a circuit that works for unsigned numbers.

A.2-6 STD_ULOGIC TYPE

In this book we use the STD_LOGIC type in most examples of VHDL code. This type is actually a *subtype* of the STD_ULOGIC type. Signals that have the STD_ULOGIC type can take the same values as the STD_LOGIC signals that we have been using. The only difference between STD_ULOGIC and STD_LOGIC has to do with the concept of a *resolution function*. In VHDL a resolution function is used to determine what value a signal should take if there are two sources for that signal. For example, two tri-state drivers could both have their outputs connected to a signal, *x*. At some given time one driver might produce the output value 'Z' and the other driver might produce the value 1. A resolution function is used to determine that the value of *x* should be 1 in this case. The STD_LOGIC type allows multiple sources for a signal; it resolves the correct value using a resolution function that is provided as part of the *std_logic_1164* package. The STD_ULOGIC type does not permit signals to have multiple sources. We have introduced STD_ULOGIC for completeness only; it is not used in this book.

A.2.7 SIGNED AND UNSIGNED TYPES

The *std_logic_signed* and *std_logic_unsigned* packages mentioned in Section A.2.5 make use of another package, called *std_logic_arith*. This package defines the type of circuit that should be used to implement the arithmetic operators, such as $+$. The *std_logic_arith* package defines two signal types, SIGNED and UNSIGNED. These types are identical to the STD_LOGIC_VECTOR type because they represent an array of STD_LOGIC signals. The purpose of the SIGNED and UNSIGNED types is to allow the user to indicate in the VHDL code what kind of number representation is being used. The SIGNED type is used in code for circuits that deal with signed (2's complement) numbers, and the UNSIGNED type is used in code that deals with unsigned numbers.

Example A.4

Assume that *A* and *B* are signals with the SIGNED type. Assume that *A* is assigned the value "1000", and *B* is assigned the value "0001". VHDL provides relational operators (see Table A.1 in Section A.3) that can be used to compare the values of two signals. The comparison $A < B$ evaluates to true because the signed values are $A = -8$ and $B=1$. On the other hand, if *A* and *B* are defined with the UNSIGNED type, then $A < B$ evaluates to false because the unsigned values are $A = 8$ and $B = 1$.

The *std_logic_signed* package specifies that STD_LOGIC_VECTOR signals should be treated like SIGNED signals. Similarly, the *std_logic_unsigned* package specifies that STD_ LOGIC_VECTOR signals should be treated like UNSIGNED signals. It is an arbitrary choice whether code is written using STD_LOGIC_VECTOR signals in conjunction with the *std_logic_signed* or *std_logic_unsigned* packages or using SIGNED and UNSIGNED signals with the *std_logic_arith* package.

A.2.8 INTEGER TYPE

The VHDL standard defines the INTEGER type for use with arithmetic operators. In this book the STD_LOGIC_VECTOR type is usually preferred in code for arithmetic circuits, but the INTEGER type is used occasionally. An INTEGER signal represents a binary number. The code does not specifically give the number of bits in the signal, as it does for STD_LOGIC_VECTOR signals. By default, an INTEGER signal has 32 bits and can represent numbers from $-(2^{31} - 1)$ to $2^{31} - 1$. This is one number less than the normal 2's complement range; the reason is simply that the VHDL standard specifies an equal number of negative and positive numbers. Integers with fewer bits can also be declared, using the RANGE keyword.

Example A.5

```
SIGNAL X : INTEGER RANGE –127 TO 127 ;
```

This statement defines *X* as an eight-bit signed number.

A.2.9 BOOLEAN TYPE

An object of type BOOLEAN can have the values TRUE or FALSE, where TRUE is equivalent to 1 and FALSE is 0.

Example A.6

```
SIGNAL Flag : Boolean ;
```

A.2.10 ENUMERATION TYPE

A SIGNAL of ENUMERATION type is one for which the possible values that the signal can have are user specified. The general form of an ENUMERATION type is

```
TYPE enumerated_type_name IS (name {, name}) ;
```

The curly brackets indicate that one or more additional items can be included. We use these brackets in this manner in several places in the appendix. The most common example of using the ENUMERATION type is for specifying the states for a finite-state machine.

Example A.7

```
TYPE State_type IS (stateA, stateB, stateC) ;
SIGNAL y : State_type ;
```

This code declares the signal *y*, for which the legal values are *stateA*, *stateB*, and *stateC*. The VHDL compiler assigns bit patterns (codes) to represent *stateA*, *stateB*, and *stateC*.

A.2.11 CONSTANT DATA OBJECTS

A CONSTANT is a data object whose value cannot be changed. Unlike a SIGNAL, a CONSTANT does not represent a wire in a circuit. The general form of a CONSTANT declaration is

```
CONSTANT constant_name : type_name := constant_value ;
```

The purpose of a constant is to improve the readability of code, by using the name of the constant in place of a value or number.

Example A.8

```
CONSTANT Zero : STD_LOGIC_VECTOR (3 DOWNTO 0) := "0000" ;
```

Then the word *Zero* can be used in the code to indicate the constant value "0000".

A.2.12 VARIABLE DATA OBJECTS

A VARIABLE, unlike a SIGNAL, does not necessarily represent a wire in a circuit. VARIABLE data objects are sometimes used to hold the results of computations and for the index variables in loops. We will give some examples in Section A.9.7.

A.2.13 TYPE CONVERSION

VHDL is a strongly type-checked language, which means that it does not permit the assignment of the value of a signal of one type to another signal that has a different type. Even for signals that intuitively seem compatible, such as BIT and STD_LOGIC, using the two types together is not permitted. To avoid this problem, we generally use only the STD_LOGIC and STD_LOGIC_VECTOR types in this book. When it is necessary to use code that has a mixture of types, type-conversion functions can be used to convert from one type to another.

Assume that *X* is defined as an eight-bit STD_LOGIC_VECTOR signal and *Y* is an INTEGER signal defined with the range from 0 to 255. An example of a conversion function that allows the value of *Y* to be assigned to *X* is

```
X <= CONV_STD_LOGIC_VECTOR(Y, 8) ;
```

This conversion function has two parameters: the name of the signal to be converted and the number of bits in *X*. The function is provided as part of the *std_logic_arith* package; hence that package must be included in the code using the appropriate LIBRARY and USE clauses.

A.2.14 ARRAYS

We said above that the BIT_VECTOR and STD_LOGIC_VECTOR types are arrays of BIT and STD_LOGIC signals, respectively. The definitions of these arrays, which are provided as part of the VHDL standards, are

```
TYPE BIT_VECTOR IS ARRAY (NATURAL RANGE <>) OF BIT ;
TYPE STD_LOGIC_VECTOR IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC ;
```

The sizes of the arrays are not set in the definitions; the syntax (NATURAL RANGE <=) has the effect of allowing the user to set the size of the array when a data object of either type is declared. Arrays of any type can be defined by the user. For example

```
TYPE Byte IS ARRAY (7 DOWNTO 0) OF STD_LOGIC ;
SIGNAL X : Byte ;
```

declares the signal *X* with the type *Byte*, which is an eight-element array of STD_LOGIC data objects.

An example that defines a two-dimensional array is

```
TYPE RegArray IS ARRAY(3 DOWNTO 0) OF STD_LOGIC_VECTOR(7 DOWNTO 0); SIGNAL R: RegArray;
```

This code defines *R* as an array with four elements. Each element is an eight-bit STD_LOGIC_VECTOR signal. The syntax *R*(*i*), where $3 \geq i \geq 0$, is used to refer to element *i* of the array. The syntax *R*(*i*)(*j*), where $7 \geq j \geq 0$, is used to refer to one bit in the array *R*(*i*). This bit has the type STD_LOGIC. An example using the *RegArray* type is given in Section 7.7.

A.3 OPERATORS

VHDL provides a number of operators that are useful for synthesizing, simulating, and documenting logic circuits. In Section 4.6.8 we discussed the operators that are used for synthesis purposes. We listed them according to their functionality. The VHDL Standard groups all operators into formal classes as shown in Table A.1. Operators in a given class have the same precedence. The precedence of classes is indicated in the table. Observe that the NOT operator is in the Miscellaneous class rather than Logical class. Hence, NOT has higher precedence than AND and OR.

In a logic expression, the operators of the same class are evaluated from left to right. Parentheses should always be used to ensure the correct interpretation of the expression. For example, the expression

```
x1 AND x2 OR x3 AND x4
```

Table A-1 The VHDL operators.

	Operator Class	Operator
Highest precedence	Miscellaneous	**, ABS, NOT
	Multiplying	*, /, MOD, REM
	Sign	+, –
	Adding	+, –, &
	Relational	=, /=, <, <=, >, >=
Lowest precedence	Logical	AND, OR, NAND, NOR, XOR, XNOR

does not have the $x_1x_2 + x_3x_4$ meaning that would be expected because AND does not have precedence over OR. To have the desired meaning, it must be written as

```
(x1 AND x2) OR (x3 AND x4)
```

A.4 VHDL Design Entity

A circuit or subcircuit described with VHDL code is called a *design entity*, or just entity. Figure A.1 shows the general structure of an entity. It has two

main parts: the entity declaration, which specifies the input and output signals for the entity, and the architecture, which gives the circuit details.

A.4.1 ENTITY DECLARATION

The input and output signals in an entity are specified using the ENTITY declaration, as indicated in Figure A.2. The name of the entity can be any legal VHDL name. The square brackets indicate an optional item. The input and output signals are specified using the keyword PORT. Whether each port is an input, output, or bidirectional signal is specified by the mode of the port. The available modes are summarized in Table A.2. If the mode of a port is not specified, it is assumed to have the mode IN.

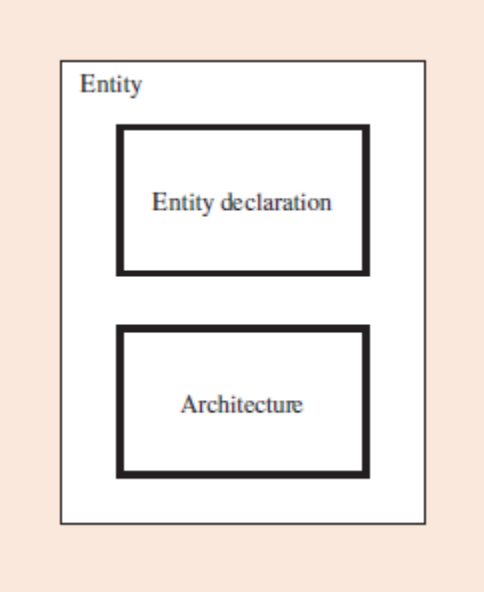


Figure A.1 The general structure of a VHDL design entity.

```
ENTITY entity_name IS
    PORT ( [SIGNAL] signal_name {, signal_name}   : [mode] type_name {;
           [SIGNAL] signal_name {, signal_name}   : [mode] type_name } ) ;
END entity_name ;
```

Figure A.2 The general form of an entity.

Table A-2 The possible modes for signals that are entity ports.

Mode	Purpose
IN	Used for a signal that is an input to an entity.
OUT	Used for a signal that is an output from an entity. The value of the signal can not be used inside the entity. This means that in an assignment statement, the signal can appear only to the left of the <= operator.
INOUT	Used for a signal that is both an input to an entity and an output from the entity.
BUFFER	Used for a signal that is an output from an entity. The value of the signal can be used inside the entity, which means that in an assignment statement, the signal can appear both on the left and right sides of the <= operator.

A.4.2 ARCHITECTURE

An ARCHITECTURE provides the circuit details for an entity. The general structure of an architecture is shown in Figure A.3. It has two main parts: the declarative region and the architecture body. The declarative region appears preceding the BEGIN keyword. It can be used to declare signals, user-defined types, and constants. It can also be used to declare components and to specify attributes; we discuss the COMPONENT and ATTRIBUTE keywords in Sections A.6 and A.10.13, respectively.

```
ARCHITECTURE architecture_name OF entity_name IS
    [SIGNAL declarations]
    [CONSTANT declarations]
    [TYPE declarations]
    [COMPONENT declarations]
    [ATTRIBUTE specifications]
BEGIN
    {COMPONENT instantiation statement ;}
    {CONCURRENT ASSIGNMENT statement ;}
    {PROCESS statement ;}
    {GENERATE statement ;}
END [architecture_name] ;
```

Figure A.3 The general form of an architecture.

The functionality of the entity is specified in the architecture body, which follows the BEGIN keyword. This specification involves statements that define the logic functions in the circuit, which can be given in a variety of ways.Wewill discuss a number of possibilities in the sections that follow.

Example A.9

Figure A.4 gives the VHDL code for an entity named fulladd, which represents a full-adder Example A.9 circuit. (The full-adder is discussed in Section 3.2.) The entity declaration specifies the input and output signals. The input port *Cin* is the carry-in, and the bits to be added are the input ports *x* and *y*. The output ports are the sum, *s*, and the carry-out, *Cout*. The input and output signals are called the *ports* of the entity. This term is adopted from the electrical jargon in which it refers to an input or output connection in an electrical circuit.

The architecture defines the functionality of the full-adder using logic equations. The name of the architecture can be any legal VHDL name. We chose the name LogicFunc for this simple example. In terms of the general form of the architecture in Figure A.3, a logic equation is a type of concurrent assignment statement. These statements are described in Section A.7.

A second example of an entity is given in Figure A.5, which gives the code for a four-bit ripple-carry adder. The *X* and *Y* inputs, as well as the output sum *S* are declared as four-bit vectors. Three signals, *C1*, *C2*, and *C3* are declared for use as internal carry signals. These signals are declared in the architecture, rather than in the entity declaration, because they are used internally in the circuit and are not input or output ports.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT ( Cin, x, y   : IN    STD_LOGIC ;
           s, Cout     : OUT   STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
    s <= x XOR y XOR Cin ;
    Cout <= (x AND y) OR (x AND Cin) OR (y AND Cin) ;
END LogicFunc ;
```

Figure A.4 Code for a full-adder.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder4 IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE LogicFunc OF adder4 IS
    SIGNAL C1, C2, C3 : STD_LOGIC;
BEGIN
    S(0) <= X(0) XOR Y(0) XOR Cin ;
    C1 <= (X(0) AND Y(0)) OR (X(0) AND Cin) OR (Y(0) AND Cin) ;
    S(1) <= X(1) XOR Y(1) XOR C1 ;
    C2 <= (X(1) AND Y(1)) OR (X(1) AND C1) OR (Y(1) AND C1) ;
    S(2) <= X(2) XOR Y(2) XOR C2 ;
    C3 <= (X(2) AND Y(2)) OR (X(2) AND C2) OR (Y(2) AND C2) ;
    S(3) <= X(3) XOR Y(3) XOR C3 ;
    Cout <= (X(3) AND Y(3)) OR (X(3) AND C3) OR (Y(3) AND C3) ;
END LogicFunc ;
```

Figure A.5 Code for a four-bit adder.

A.5 PACKAGE

A VHDL package serves as a repository. It is used to hold VHDL code that is of general use, like the code that defines a type. The package can be included for use in any number of other source-code files, which can then use the definitions provided in the package. Like an architecture, introduced in Section A.4.2, a package can have two main parts: the *package declaration* and the *package_body*. The package body is an optional part, which we do not use in this book; one use of a package body is to define VHDL functions, such as the conversion functions introduced in Section A.2.13.

The general form of a package declaration is depicted in Figure A.6. Definitions provided in the package, such as the definition of a type, can be used in any source-code file that includes the statements

```
LIBRARY library_name ;
USE library_name.package_name.all ;
```

The *library_name* represents the location in the computer file system where the package is stored. A library can either be provided as part of a CAD system, in which case it is termed a *system library*, or be created by the user, in which case it is called a *user library*. An example of a system library is the ieee library. We discussed four packages in that library in Section A.2: *std_logic_1164*, *std_logic_signed*, *std_logic_unsigned*, and *std_logic_arith*.

```
PACKAGE package_name IS
    [TYPE declarations]
    [SIGNAL declarations]
    [COMPONENT declarations]
END package_name ;
```

Figure A.6 The general form of a PACKAGE declaration.

A special case of a user library is represented by the file-system directory where the VHDL source-code file that declares a package is stored. This directory can be referred to by the library name *work*, which stands for *working directory*. Hence, if a source-code afile that contains a package declaration called *user_package_name* is compiled, then the package can be used in another source-code file (which is stored in the same file-system directory) by including the statements

```
LIBRARY work ;
USE work.user_package_name.all ;
```

Actually, for the special case of the *work* library, the LIBRARY clause is not required, because the work library is always accessible.

Figure A.6 shows that the package declaration can be used to declare signals and components. Components are discussed in the next section. A signal declared in a package can be used by any design entity that accesses the package. Such signals are similar in concept to global variables used in computer programming languages. In contrast, a signal declared in an architecture can be used only inside that architecture. Such signals are analogous to local variables in a programming language.

A.6 Using Subcircuits

AVHDL entity defined in one source-code file can be used as a subcircuit in another sourcecode file. In VHDL jargon the subcircuit is called a *component*. A subcircuit must be declared using a *component declaration*. This statement specifies the name of the subcircuit and gives the names of its input and output ports. The component declaration can appear either in the declaration region of an architecture or in a package declaration. The general form of the statement is shown in Figure A.7. The syntax used is similar to the syntax in an entity declaration.

Once a component declaration is given, the component can be *instantiated* as a subcircuit. This is done using a *component instantiation* statement. It has the general form

```
instance_name : component_name PORT MAP (
    [formal_name =>] actual_name {, [formal_name =>] actual_name} ) ;

COMPONENT component_name
      [GENERIC ( parameter_name : integer := default_value {;
                  parameter_name : integer := default_value} ) ;]
      PORT ( [SIGNAL] signal_name {, signal_name}  : [mode] type_name {;
             [SIGNAL] signal_name {, signal_name}  : [mode] type_name } ) ;
END COMPONENT ;
```

Figure A.7 The general form of a component declaration.

Each *formal_name* is the name of a port in the subcircuit. Each *actual_name* is the name of a signal in the code that instantiates the subcircuit. The syntax “formal name =>” is provided so that the order of the signals listed after the PORT MAP keywords does not have to be the same as the order of the ports in the corresponding COMPONENT declaration. In VHDL jargon this is called the *named association*. If the signal names following the PORT MAP keywords are given in the same order as in the COMPONENT declaration, then “formal_name => is not needed. This is called the *positional association*.

An example using a component (subcircuit) is shown in Figure A.8. It gives the code for a four-bit ripple-carry adder built using four instances of the *fulladd* subcircuit. The
inputs to the adder are the carry-in, *Cin*, and the 2 four-bit numbers *X* and *Y*. The output is the four-bit sum, *S*, and the carry-out, *Cout*. We have chosen the name Structure in the architecture because the hierarchical style of code

that uses subcircuits is often called the *structural* style. A three-bit signal, C, is declared to represent the carry-outs from stages 0, 1, and 2.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
    COMPONENT fulladd
        PORT ( Cin, x, y : IN   STD_LOGIC ;
               s, Cout   : OUT  STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP (
        x => X(3), y => Y(3), Cin => C(3), s => S(3), Cout => Cout ) ;
END Structure ;
```

Figure A.8 Code for a four-bit adder, using component instantiation.

The next statement in the architecture gives the component declaration for the *fulladd* subcircuit. The architecture body instantiates four copies of the full-adder subcircuit. In the first three instantiation statements, we have used positional association because the signals are listed in the same order as given in the declaration for the fulladd component in Figure A.4 The last instantiation statement gives an example of named association. Note that it is legal to use the same name for a signal in the architecture that is used for a port name in a component. An example of this is the *Cout* signal. The signal names used in the instantiation statements implicitly specify how the component instances are interconnected to create the adder entity.

A.6.1 DECLARING A COMPONENT IN A PACKAGE

Figure A.6 shows that a component declaration can be given in a package. An example is shown in Figure A.9. It defines the package named

fulladd_package, which provides the component declaration for the *fulladd* entity. This package can be stored in a separate source-code file or can be included at the end of the file that defines the *fulladd* entity (see Figure A.4). Any source-code that includes the statement "USE work.fulladd_package.all" can use the *fulladd* component as a subcircuit. Figure A.10 shows how a four-bit ripplecarry adder entity can be written to use the package. The code is the same as that in Figure A.8 except that it includes the extra USE clause for the package and deletes the component declaration statement from the architecture.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE fulladd_package IS
    COMPONENT fulladd
        PORT ( Cin, x, y  : IN   STD_LOGIC ;
               s, Cout    : OUT  STD_LOGIC ) ;
    END COMPONENT ;
END fulladd_package ;
```

Figure A.9 An example of a package declaration.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), Y(3), S(3), Cout ) ;
END Structure ;
```

Figure A.10 Using a component defined in a package.

A.7 CONCURRENT ASSIGNMENT STATEMENTS

A concurrent assignment statement is used to assign a value to a signal in an architecture body. An example was given in Figure A.4, in which the logic expressions illustrate one type of concurrent assignment statement. VHDL provides four different types of concurrent assignment statements: simple signal assignment, selected signal assignment, conditional signal assignment, and generate statements.

A.7.1 SIMPLE SIGNAL ASSIGNMENT

A simple signal assignment statement is used for a logic or an arithmetic expression. The general form is

```
signal_name <= expression ;
```

where <= is the VHDL *assignment operator.* The following examples illustrate its use.

```
SIGNAL x1, x2, x3, f  : STD_LOGIC ;
      .
      .
      .
f <= (x1 AND x2) OR x3 ;
```

This code defines f in a logic expression, which involves single-bit quantities. VHDL also supports multibit logic expressions, as in

```
SIGNAL A, B, C  : STD_LOGIC_VECTOR (1 TO 3) ;
      .
      .
      .
C <= A AND B ;
```

This results in $C(1) = A(1) \cdot B(1)$, $C(2) = A(2) \cdot B(2)$, and $C(3) = A(3) \cdot B(3)$.

An example of an arithmetic expression is

```
SIGNAL X, Y, S   : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
      .
      .
      .
S <= X + Y ;
```

This represents a four-bit adder, without carry-in and carry-out. We can alternatively declare a carry-in signal, *Cin*, and a five-bit signal, *Sum*, as follows

```
SIGNAL Cin   : STD_LOGIC ;
SIGNAL Sum   : STD_LOGIC_VECTOR (4 DOWNTO 0) ;
```

Then the statement

```
Sum <= ('0' & X) + ('0' & Y) + Cin ;
```

represents the four-bit adder with carry-in and carry-out. The four sum bits are *Sum*(3) to *Sum*(0), while the carry-out is the bit *Sum*(4). The syntax ('0' & X) uses the VHDL *concatenate operator,* &, to put a 0 on the left end of the signal *X*. The reader should not confuse this use of the & symbol with the logical AND operation, which is the usual meaning of this symbol; in VHDL the logical AND is indicated by the word AND, and & means concatenate. The concatenate operation prepends a 0 digit onto X, creating a five-bit number. VHDL requires at least one of the operands of an arithmetic expression to have the same number of bits as the signal used to hold the result. The complete code for the four-bit adder with carry signals is given in Figure A.11. We should note that this is a different way (it is actually a better way) to describe a four-bit adder, in comparison with the structural code in Figure A.8. Observe that the statement "S <= Sum(3 DOWNTO 0)" assigns the lower four bits of the *Sum* signal, which are the four sum bits, to the output *S*.

A.7.2 ASSIGNING SIGNAL VALUES USING OTHERS

Assume that we wish to set all bits in the signal *S* to 0. As we already know, one way to do so is to write "S <= "0000" ;". If the number of bits in *S* is large, a more convenient way of expressing the assignment statement is to use the OTHERS keyword, as in

```
        S <= (OTHERS => '0' ) ;

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder ;

ARCHITECTURE Behavior OF adder IS
    SIGNAL Sum : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + ('0' & Y) + Cin ;
    S <= Sum(3 DOWNTO 0) ;
    Cout <= Sum(4) ;
END Behavior ;
```

Figure A.11 Code for a four-bit adder, using arithmetic expressions.

This statement also sets all bits in *S* to 0. But it has the benefit of working for any number of bits, not just four. In general, the meaning of (OTHERS => *Value)* is to set each bit of the destination operand to *Value.* An example of code that uses this construct is shown in Figure A.28.

A.7.3 SELECTED SIGNAL ASSIGNMENT

A selected signal assignment statement is used to set the value of a signal to one of several alternatives based on a selection criterion. The general form is

```
[label:] -- an optional label can be placed here
WITH expression SELECT
    signal_name <=  expression WHEN constant_value{,
                    expression WHEN constant_value} ;
```

Example A.10

The code below describes a 2-to-1 multiplexer with *Sel* as the select input. In a selected signal assignment, all possible values of the select input, *Sel* in this case, must be explicitly listed in the code. The word OTHERS provides

an easy way to meet this requirement. OTHERS represents all possible values not already listed. In this case the other possible values are 1, Z, –, and so on. Another requirement for the selected signal assignment is that each WHEN clause must specify a criterion that is mutually exclusive of the criteria in all other WHEN clauses.

```
SIGNAL x1, x2, Sel, f : STD_LOGIC ;
        .
        .
        .
WITH Sel SELECT
    f <= x1 WHEN '0' ,
         x2 WHEN OTHERS ;
```

A.7.4 CONDITIONAL SIGNAL ASSIGNMENT

Similar to the selected signal assignment, the conditional signal assignment is used to set a signal to one of several alternative values. The general form is

```
[label:]
signal_name <= expression WHEN logical_condition ELSE
               {expression WHEN logical_condition ELSE}
               expression ;
```

An example is

```
f <= '1' WHEN x1 = x2 ELSE '0' ;
```

One key difference in comparison with the selected signal assignment has to be noted. The logical conditions listed after each WHEN clause need not be mutually exclusive, because the conditions are given a priority from the first listed to the last listed. This is illustrated by the example in Figure A.12. The code represents a priority encoder in which the highest-priority request is indicated as the output of the circuit. (Encoder circuits are described in Chapter 4.) The output, /, of the priority encoder comprises two bits whose values depend on the three inputs, *req1*, *req2*, and *req3*. If *req1* is 1, then *f* is set to 01. If *req2* is 1, then *f* is set to 10, but only if req1 is not also 1. Hence req1 has higher priority than req2. Similarly, req1 and req2 have higher priority than req3. Thus if req3 is 1, then *f* is 11, but only if

neither *req*1 nor *req*2 is also 1. For this priority encoder, if none of the three inputs is 1, then *f*is assigned the value 00.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( req1, req2, req3  : IN   STD_LOGIC ;
           f                 : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    f <= "01" WHEN req1 = '1' ELSE
         "10" WHEN req2 = '1' ELSE
         "11" WHEN req3 = '1' ELSE
         "00" ;
END Behavior;
```

Figure A.12 A priority encoder described with a conditional signal assignment.

```
generate_label:
FOR index_variable IN range GENERATE
    statement ;
    {statement ;}
END GENERATE ;


generate_label:
IF expression GENERATE
    statement ;
    {statement ;}
END GENERATE ;
```

Figure A.13 The general forms of the GENERATE statement.

A.7.5 GENERATE STATEMENT

There are two variants of the GENERATE statement: the FOR GENERATE and the IF GENERATE. The general form of both types is shown in Figure A.13. The IF GENERATE statement is seldom needed, but FOR GENERATE is often used in practice. It provides a convenient way of repeating either a logic expression or a component instantiation. Figure A.14 illustrates its use for component instantiation. The code in the figure is equivalent to the code given in Figure A.8.

A.8 DEFINING AN ENTITY WITH GENERICS

The code in Figure A.14 represents an adder for four-bit numbers. It is possible to make this code more general by introducing a parameter in the code that represents the number of bits in the adder. In VHDL jargon such a parameter is called a GENERIC. Part (a)of

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(0 TO 4) ;
BEGIN
    C(0) <= Cin ;
    Generate_label:
    FOR i IN 0 TO 3 GENERATE
        bit: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
    END GENERATE ;
    Cout <= C(4) ;
END Structure ;
```

Figure A.14 An example of component instantiation with FOR GENERATE.

Figure A.15 augments the code from Figure A.14 to describe an «-bit adder entity, named *addern.* The GENERIC keyword is used to define the number of bits, n, to be added. This parameter is used in the code, both in the definitions of the signals *X*, *Y*, and *S* and in the FOR GENERATE statement that instantiates the *n* full-adders.

Part (*b*) of Figure A.15 shows how the adder described with arithmetic assignment statements in Figure A.11 can be modified to represent an n-bit adder. The *Sum* signal is declared to have $n + 1$ bits. The lower *n* bits of this signal provide the sum output *S* of the adder, and the most-significant bit *Sum(n)* gives the carry-out.

It is possible to use the GENERIC feature with components that are instantiated as subcircuits in other code. In Section A.10.9 we give an

example that uses the *addern* entity as a subcircuit.

A.9 SEQUENTIAL ASSIGNMENT STATEMENTS

The order in which the concurrent assignment statements in an architecture body appear does not affect the meaning of the code. Many types of logic circuits can be described using these statements. However, VHDL also provides another type of statements, called *sequential assignment statements*, for which the order of the statements in the code can affect the semantics of the code. There are three variants of the sequential assignment statements: IF statement, CASE statement, and LOOP statements.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY addern IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( Cin    : IN    STD_LOGIC ;
           X, Y   : IN    STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
           S      : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
           Cout   : OUT STD_LOGIC ) ;
END addern ;

ARCHITECTURE Structure OF addern IS
    SIGNAL C : STD_LOGIC_VECTOR(0 TO n) ;
BEGIN
    C(0) <= Cin ;
    Generate_label:
    FOR i IN 0 TO n-1 GENERATE
        stage: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
    END GENERATE ;
    Cout <= C(4) ;
END Structure ;
```

Figure A.15 A *n*-bit adder (Part *a*).

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY addern IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END addern ;

ARCHITECTURE Behavior OF addern IS
    SIGNAL Sum : STD_LOGIC_VECTOR(n DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + ('0' & Y) + Cin ;
    S <= Sum(n-1 DOWNTO 0) ;
    Cout <= Sum(n) ;
END Behavior ;
```

Figure A.15 A *n*-bit adder (Part *b*).

```
[process_label:]
PROCESS [( signal name {, signal name} )]
    [VARIABLE declarations]
BEGIN
    [WAIT statement]
    [Simple Signal Assignment Statements]
    [Variable Assignment Statements]
    [IF Statements]
    [CASE Statements]
    [LOOP Statements]
END PROCESS [process_label] ;
```

Figure A.16 The general form of a PROCESS statement.

A.9.1 PROCESS STATEMENT

Since the order in which the sequential statements appear in VHDL code is significant, whereas the ordering of concurrent statements is not, the sequential statements must be separated from the concurrent statements. This is accomplished using a PROCESS statement. The PROCESS statement appears inside an architecture body, and it encloses other statements within it. The IF, CASE, and LOOP statements can appear only inside a process. The general form of a PROCESS statement is shown in Figure A.16. Its structure is somewhat similar to an architecture. VARIABLE data objects can be declared (only) inside the process. Any

variable declared can be used only by the code within the process; we say that the *scope* of the variable is limited to the process. To use the value of such a variable outside the process, the variable's value can be assigned to a signal. The various elements of the process are best explained by giving some examples. But first we need to introduce the IF, CASE, and LOOP statements.

The IF, CASE, and LOOP statements can be used to describe either combinational or sequential circuits. We will introduce these statements by giving some examples of combinational circuits because they are easier to understand. Sequential circuits are described in Section A.10.

A.9.2 IF STATEMENT

The general form of an IF statement is given in Figure A.17. An example using an IF statement for combinational logic is

```
IF Sel = '0' THEN
    f <= x1 ;
ELSE
    f <= x2 ;
END IF ;
```

```
IF expression THEN
    statement ;
    {statement ;}
ELSIF expression THEN
    statement ;
    {statement ;}
ELSE
    statement ;
    {statement ;}
END IF ;
```

Figure A.17 The general form of an IF statement.

This code defines a 2-to-1 multiplexer. Examples of sequential logic described with IF statements are given in Section A.10.

A.9.3 CASE STATEMENT

The general form of a CASE statement is shown in Figure A.18. The *constant value* can be a single value, such as 2, a list of values separated by

the | pipe, such as 2|3, or a range, such as 2 to 4. An example of a CASE statement used to describe combinational logic is

```
CASE Sel IS
    WHEN '0' =>
        f <= x1 ;
    WHEN OTHERS =>
        f <= x2 ;
END CASE ;
```

This code represents the same 2-to-1 multiplexer described in Section A.9.2 using the IF statement. Similar to a selected signal assignment, all possible valuations of the expression

```
CASE expression IS
    WHEN constant_value =>
        statement ;
        {statement ;}
    WHEN constant_value =>
        statement ;
        {statement ;}
    WHEN OTHERS =>
        statement ;
        {statement ;}
END CASE ;
```

Figure A.18 The general form of a CASE statement.

```
[loop_label:]
FOR variable_name IN range LOOP
    statement ;
    {statement ;}
END LOOP [loop_label] ;


[loop_label:]
WHILE boolean_expression LOOP
    statement ;
    {statement ;}
END LOOP [loop_label] ;
```

Figure A.19 The general forms of FOR-LOOP and WHILE-LOOP statements.

used for the WHEN clauses must be listed; hence the OTHERS keyword is needed. Also, all WHEN clauses in the CASE statement must be mutually exclusive. Examples of sequential circuits described with the CASE statement are given in Section A.10.10.

A.9.4 LOOP STATEMENTS

VHDL provides two types of loop statements: the FOR-LOOP statement and the WHILE-LOOP statement. Their general forms are shown in Figure A.19. These statements are used to repeat one or more sequential assignment statements in much the same way as a FOR GENERATE statement is used to repeat concurrent assignment statements. Examples of the FOR-LOOP are given in Section A.9.7.

A.9.5 USING A PROCESS FOR A COMBINATIONAL CIRCUIT

An example of a PROCESS statement is shown in Figure A.20. It includes the code for the IF statement from Section A.9.2. The signals *Sel*, *x1*, and *x2* are shown in parentheses after the PROCESS keyword. They indicate which signals the process depends on and are called the *sensitivity list* of the process. For a process that describes combinational logic, as in this example, the sensitivity list includes all input signals used inside the process.

```
PROCESS ( Sel, x1, x2 )
BEGIN
    IF Sel = '0' THEN
        f <= x1 ;
    ELSE
        f <= x2 ;
    END IF ;
END PROCESS ;
```

Figure A.20 A PROCESS statement.

In VHDL jargon a process is described as follows. When the value of a signal in the sensitivity list changes, the process becomes *active*. Once active, the statements inside the process are "evaluated" in sequential order. Any signal assignments made in the process take effect only after all the statements inside the process have been evaluated. We say that the signal assignments performed inside the process are *scheduled* and will take effect at the end of the process.

The process describes a logic circuit and is translated into logic expressions in the same manner as the concurrent assignment statements in an architecture body. The concept of the process statements being evaluated in sequence provides a convenient way of understanding the semantics of

the code inside a process. In particular, a key concept is that if multiple assignments are made to a signal inside a process, only the last one to be evaluated has any effect. This is illustrated in the next example.

A.9.6 STATEMENT ORDERING

The IF statement in Figure A.20 describes a multiplexer that assigns either of two inputs, *x*1 or *x*2, to the output *f*. Another way of describing the multiplexer with an IF statement is shown in Figure A.21. The statement "f <= xl ;" is evaluated first. However, the signal /may not actually be changed to the value of *x*l, because there may be a subsequent assignment to *f* in the code inside the process statement. At this point in the process, xl represents the *default* value for *f* if no other assignment to *f* is evaluated. If we assume that *Sel* = 1, then the statement "f <= x2 ;" will be evaluated. The effect of this second assignment to *f* is to override the default assignment. Hence the result of the process is that *f*is set to the value *x*2 when *Sel* = l. If we assume that *Sel* = 0, then the IF condition fails and *f* is assigned its default value, *x*l.

This example illustrates the effect of the ordering of statements inside a process. If the two statements were reversed in order, then the IF statement would be evaluated first and the statement "f <= xl ;" would be evaluated last. Hence the process would always result in *f*being set to the value of *x*l.

Implied Memory

Consider the process in Figure A.22. It is the same as the process in Figure A.21 except that the default assignment statement "f <= xl ;" has been removed. Because the process does not specify a default value for f, and there is no ELSE clause in the IF statement, the meaning of the process is that *f*should retain its present value when the IF condition is not

```
PROCESS ( Sel, x1, x2 )
BEGIN
    f <= x1 ;
    IF Sel = 1 THEN
        f <= x2 ;
    END IF ;
END PROCESS ;
```

Figure A.21 An example illustrating the ordering of statements within a PROCESS.

```
PROCESS ( Sel, x2 )
BEGIN
    IF Sel = 1 THEN
        f <= x2 ;
    END IF ;
END PROCESS ;
```

Figure A.22 An example of implied memory.

satisfied. The following expression is generated by the VHDL compiler for this process

$$f = Sel \cdot x2 + \overline{Sel} \cdot f$$

Hence when *Sel* = 0, the value of *x2* is "remembered" at the outputf. In VHDL jargon this is called *implied memory* or *implicit memory.* Although it is rarely useful for combinational circuits, we will show shortly that implied memory is the key concept used to describe sequential circuits.

A.9.7 USING A VARIABLE IN A PROCESS

We mentioned earlier that VHDL provides VARIABLE data objects, in addition to SIGNAL data objects. Unlike a signal, a variable data object does not represent a wire in a circuit. Therefore, a variable can be used to describe the functionality of a logic circuit in ways that are not possible using a signal. This concept is illustrated in Figure A.23. The intent of the code is to describe a logic circuit that counts the number of bits in the three-bit signal *X* that are equal to 1. The count is output using the signal called Count,which is a two-bit unsigned integer. Notice that *Count* is declared with the mode *Buffer* because it is used in the architecture body on both the left and right sides of an assignment operator. Table A.2 explains the meaning of the *Buffer* mode.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY numbits IS
    PORT ( X       : IN     STD_LOGIC_VECTOR(1 TO 3) ;
           Count   : BUFFER INTEGER RANGE 0 TO 3 ) ;
END numbits ;

ARCHITECTURE Behavior OF numbits IS
BEGIN
    PROCESS ( X )   - - count the number of bits in X with the value 1
    BEGIN
        Count <= 0 ;    - - the 0 with no quotes is a decimal number
        FOR i IN 1 TO 3 LOOP
            IF X(i) = '1' THEN
                Count <= Count + 1 ;
            END IF ;
        END LOOP ;
    END PROCESS ;
END Behavior ;
```

Figure A.23 A FOR-LOOP that does not represent a sensible circuit.

Inside the process, *Count* is initially set to 0. No quotes are used for the number 0 in this case, because VHDL allows a decimal number, which we said in Section A.2.2 is denoted with no quotes, to be assigned to an INTEGER signal. The code gives a FOR-LOOP with the loop index variable i. For the values of *i* from 1 to 3, the IF statement inside the FOR-LOOP checks the value of bit X(i); if it is 1, then the value of *Count* is incremented. The code given in the figure is legal VHDL code and can be compiled without generating any errors. However, it will not work as intended, and it does not represent a sensible logic circuit.

There are two reasons why the code in Figure A.23 will not work as intended. First, there are multiple assignment statements for the signal *Count* within the process. As explained for the previous example, only the last of these assignments will have any effect. Hence if any bit in *X* is 1, then the statement "Count <= '0' ;" will not have the desired effect of initializing *Count* to 0, because it will be overridden by the assignment statement in the FOR-LOOP. Also, the FOR-LOOP will not work as desired, because each iteration for which *X* (1) is 1 will override the effect of the previous iteration. The second reason why the code is not sensible is that the statement "Count <= Count + '1' ;" describes a circuit with

feedback. Since the circuit is combinational, such feedback will result in oscillations and the circuit will not be stable.

The desired behavior of the VHDL code in Figure A.23 can be achieved using a variable, instead of a signal. This is illustrated in Figure A.24, in which the variable *Tmp* is used instead of the signal *Count* inside the process. The value of *Tmp* is assigned to *Count* at the end of the process. Observe that the assignment statements to *Tmp* are indicated with the := operator, as opposed to the <= operator. The := is called the *variable assignment operator.* Unlike <=, it does not result in the assignment being *scheduled* until the end of the process. The variable assignment takes place immediately. This *immediate* assignment solves the first of the two problems with the code in Figure A.23. The second problem is also solved by using a variable instead of a signal. Because the variable does not represent a wire in a circuit, the FOR-LOOP need not be literally interpreted as a circuit with feedback. By using the variable, the FOR-LOOP represents only a desired *behavior*,or*functionality*,of the circuit. When the code is translated, the VHDL compiler will generate a combinational circuit that implements the functionality expressed in the FOR-LOOP.

When the code in Figure A.24 is translated by the VHDL compiler, it produces the circuit with 2 two-bit adders shown in Figure A.25. It is possible to see how this circuit corresponds to the FOR-LOOP in the code. The result of the first iteration of the loop is that *Count* is set to the value of *X*(1). The second iteration then adds *X*(1) to *X*(2). This is realized by the top adder in the figure. The third iteration adds *X* (3) to the sum produced from the second iteration. This corresponds to the bottom adder. When this circuit is optimized by the logic synthesis algorithms, the resulting expressions for *Count* are

$$\text{Count}(1) = \text{X}(1)\text{X}(2) + \text{X}(1)\text{X}(3) + \text{X}(2)\text{X}(3)$$
$$\text{Count}(0) = \text{X}(1) \oplus \text{X}(2) \oplus \text{X}(3)$$

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY Numbits IS
    PORT ( X      : IN  STD_LOGIC_VECTOR(1 TO 3) ;
           Count  : OUT INTEGER RANGE 0 TO 3 ) ;
END Numbits ;

ARCHITECTURE Behavior OF Numbits IS
BEGIN
    PROCESS ( X )   - - count the number of bits in X equal to 1
        VARIABLE Tmp : INTEGER ;
    BEGIN
        Tmp := 0 ;
        FOR i IN 1 TO 3 LOOP
            IF X(i) = '1' THEN
                Tmp := Tmp + 1 ;
            END IF ;
        END LOOP ;
        Count <= Tmp ;
    END PROCESS ;
END Behavior ;
```

Figure A.24 The FOR-LOOP from Figure A.23 using a variable.

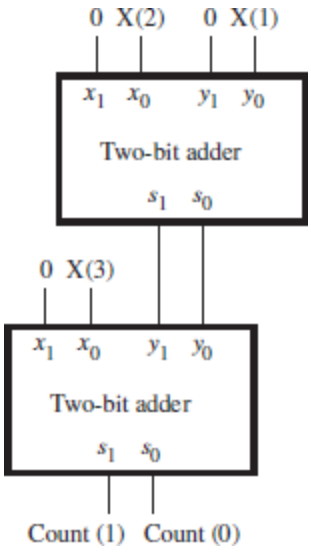


Figure A.25 The circuit generated from the code in Figure A.24.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X  : IN   STD_LOGIC_VECTOR(1 TO n) ;
           f  : OUT  STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
BEGIN
    PROCESS ( X )
        VARIABLE Tmp : STD_LOGIC ;
    BEGIN
        Tmp := X(1) ;
        AND_bits: FOR i IN 2 TO n LOOP
            Tmp := Tmp AND X(i) ;
        END LOOP AND_bits ;
        f <= NOT Tmp ;
    END PROCESS ;
END Behavior ;
```

Figure A.26 Using a variable to describe an «-input NAND gate.

These expressions represent a full-adder circuit, with *Count(0)* as the sum output and *Count(X)* as the carry-out. It is interesting to note that even though the VHDL code describes the desired behavior of the circuit in an abstract way, using a FOR-LOOP, in this example the logic synthesis algorithms produce the most efficient circuit, which is the full-adder. As we said at the beginning of this appendix and in Section 2.10, the style of code in Figure A.24 should be avoided, because it is often difficult for the designer to envisage what logic circuit the code represents.

As another example of the use of a variable, Figure A.26 gives the code for an n-bit NAND gate entity, named *NANDn.* The number of inputs to the NAND gate is set by the GENERIC parameter n. The inputs are the n-bit signal X, and the output is *f*. The variable *Tmp* is defined in the architecture and originally set to the value of the input signal *X* (1). In the FOR LOOP, *Tmp* is ANDed successively with input signals X(2) to X(n). Since *Tmp* is a variable data object, assignments to it take effect immediately; they are not scheduled to take effect at the end of the process. The complement of *Tmp* is assigned tof, thus completing the description of the *n*-input NAND operation.

Figure A.27 shows the same code given in Figure A.26 but with the data object *Tmp* defined as a signal, instead of as a variable. This code gives a

wrong result, because only the last statement included in the process has any effect on *Tmp*. The code results in $Tmp = Tmp \cdot X(4)$, as determined by the last iteration of the FOR LOOP. Also, since *Tmp* is never initialized, its value is unknown. Hence the value of the output $f = \overline{Tmp}$ is unknown.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X  : IN     STD_LOGIC_VECTOR(1 TO n) ;
           f  : OUT  STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
    SIGNAL Tmp : STD_LOGIC ;
BEGIN
    PROCESS ( X )
    BEGIN
        Tmp <= X(1) ;
        AND_bits: FOR i IN 2 TO n LOOP
            Tmp <= Tmp AND X(i) ;
        END LOOP AND_bits ;
        f <= NOT Tmp ;
    END PROCESS ;
END Behavior ;
```

Figure A.27 The code from Figure A.26 using a signal.

Figure A.28 shows one way to describe the «-input NAND gate using signals. Here *Tmp* is defined as an «-bit signal, which is set to contain « 1s using the (OTHERS =>) construct. The conditional signal assignment specifies that *f* is 0 only if all bits in the input *X* are 1, thus describing the NAND operation.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X  : IN   STD_LOGIC_VECTOR(1 TO n) ;
           f  : OUT  STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
    SIGNAL Tmp : STD_LOGIC_VECTOR(1 TO n) ;
BEGIN
    Tmp <= (OTHERS => '1') ;
    f <= '0' WHEN X = Tmp ELSE '1' ;
END Behavior ;
```

Figure A.28 Using a signal to describe an «-input NAND gate.

A final example of variables used in a sequential circuit is given in Section A.10.7. In general, using both variables and signals in VHDL code can lead to confusion because they imply different semantics. Since variables do not necessarily represent wires in a circuit, the meaning of code that uses variables is sometimes ill defined. To avoid confusion, in this book we use variables only for the loop indices in FOR GENERATE and FOR LOOP statements. Except for similar purposes, the reader should avoid using variables because they are not needed for describing logic circuits.

A.10 Sequential Circuits

Although combinational circuits can be described using either concurrent or sequential assignment statements, sequential circuits can be described only with sequential assignment statements. We now give some representative examples of sequential circuits.

A.10.1 A Gated D Latch

Figure A.29 gives the code for a gated D latch. The process sensitivity list includes both the latch's data input, D, and clock, elk. Hence whenever a change occurs in the value of either *D* or elk, the process becomes active. The IF statement specifies that Q should be set to the value of *D* whenever

the clock is 1. There is no ELSE clause in the IF statement. As we explained for Figure A.22, this implies that Q should retain its present value when the IF condition is not met.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
    PORT ( D, clk  : IN     STD_LOGIC ;
           Q       : OUT  STD_LOGIC ) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
    PROCESS ( D, clk )
    BEGIN
        IF clk = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.29 A gated D Latch.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock  : IN     STD_LOGIC ;
           Q         : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.30 D flip-flop.

A.10.2 D FLIP-FLOP

Figure A.30 gives a process that is slightly different from the one in Figure A.29. The sensitivity list includes only the *Clock* signal, which means that

the process is active only when the value of *Clock* changes. The condition in the IF statement looks unusual. The syntax Clock'EVENT represents a *change* in the value of the clock signal. In VHDL jargon 'EVENT is called an *attribute*, and combining 'EVENT with a signal name, such as *Clock*, yields a logical condition. The combination in the IF statement of the two conditions Clock'EVENT and Clock = '1' specifies that Q should be assigned the value of *D* when "a change occurs in the value of *Clock*, and *Clock* is now 1". This describes a low-to-high transition of the clock signal; hence the code describes a positive-edge-triggered D flip-flop.

The *stdJogicJ.164* package defines the two functions named *risingjedge* and *falling .edge*. They can be used as a short-form notation for the condition that checks for the occurrence of a clock edge. In Figure A.30 we could replace the line "IF Clock'EVENT AND Clock = '1' THEN" with the equivalent line "IF RISING_EDGE(Clock) THEN". We do not use *risingjedge* or *falling jedge* in this book; they are mentioned for completeness.

A.10.3 USING A WAIT UNTIL STATEMENT

The process in Figure A.31 uses a different syntax to describe a D flip-flop. Synchronization with the clock edge is specified by using the statement "WAIT UNTIL Clock'EVENT AND Clock = '1' ;". A process that uses a WAIT UNTIL statement is a special case because the sensitivity list is omitted. Use of this WAIT UNTIL statement implicitly specifies that the sensitivity list includes only *Clock*. For our purposes, which is using VHDL for synthesis of circuits, a process can include a WAIT UNTIL statement only if it is the first statement in the process.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock  : IN   STD_LOGIC ;
           Q         : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        Q <= D ;
    END PROCESS ;
END Behavior ;
```

Figure A.31 Equivalent code to Figure A.30, using a WAIT UNTIL statement.

The WAIT UNTIL statement above can be written more simply as

```
WAIT UNTIL Clock = '1' ;
```

which means "wait for the next positive edge of the *Clock* signal". But, since some CAD synthesis tools require the inclusion of the 'EVENT attribute, we include the attribute in our examples.

As seen in Figures A.30 and A.31, both IF and WAIT UNTIL statements can be used to describe flip-flops. If a process only defines flip-flops, then it makes no difference which construct is used. However, in practical designs a process often includes many statements. If one or more of these statements specify a combinational subcircuit, then it is necessary to use IF statements to infer flip-flops where desired. If the WAIT UNTIL statement is used, which has to be the first statement in the process, then there will be flip-flops inferred for all statements in the process. For this reason, designers prefer using the IF statement.

A.10.4 A FLIP-FLOP WITH ASYNCHRONOUS RESET

Figure A.32 gives a process that is similar to the one in Figure A.30. It describes a D flip-flop with an asynchronous reset, or clear, input. The reset signal has the name *Resetn*. When *Resetn* = 0, the flip-flop output Q is set to 0. Appending the letter *n* to a signal name is a widely used convention to denote an active-low signal.

A.10.5 SYNCHRONOUS RESET

Figure A.33 shows how a flip-flop with a synchronous reset input can be described by using the IF statement. Figure A.34 presents a specification based on the WAIT UNTIL statement.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.32 D flip-flop with asynchronous reset.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            IF Resetn = '0' THEN
                Q <= '0' ;
            ELSE
                Q <= D ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.33 D flip-flop with synchronous reset, using an IF statement.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSE
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.34 D flip-flop with synchronous reset, using a WAIT UNTIL statement.

A.10.6 REGISTERS

One possible approach for describing a multibit register is to create an entity that instantiates multiple flip-flops. A more convenient method is illustrated in Figure A.35. It gives the same code shown in Figure A.32 but using the four-bit STD LOGIC VECTOR input *D* and the four-bit output Q. The code describes a four-bit register with asynchronous clear.

Figure A.36 gives the code for an entity named *regn*. It shows how the code in Figure A.35 can be extended to represent an n-bit register. The number of flip-flops is set by the generic parameter *n*.

The code in Figure A.37 shows how an enable input can be added to the n-bit register from Figure A.36. When the active clock edge occurs, the flip-flops in the register cannot change their stored values if the enable *E* is 0. If $E = 1$, the register responds to the active clock edge in the normal way.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg4 IS
    PORT ( D              : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Resetn, Clock  : IN    STD_LOGIC ;
           Q              : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END reg4 ;

ARCHITECTURE Behavior OF reg4 IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= "0000" ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.35 Code for a four-bit register with asynchronous clear.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( D              : IN    STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
           Resetn, Clock  : IN    STD_LOGIC ;
           Q              : OUT   STD_LOGIC_VECTOR(n-1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.36 Code for a *n*-bit register with asynchronous clear.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regne IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( D          : IN   STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
           Resetn     : IN   STD_LOGIC ;
           E, Clock   : IN   STD_LOGIC ;
           Q          : OUT  STD_LOGIC_VECTOR(n-1 DOWNTO 0) ) ;
END regne ;

ARCHITECTURE Behavior OF regne IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            IF E = '1' THEN
                Q <= D ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.37 VHDL code for an *n*-bit register with an enable input.

A.10.7 SHIFT REGISTERS

An example of code that defines a four-bit shift register is shown in Figure A.38. The lines of code are numbered for ease of reference. The shift register has a serial input, w, and parallel outputs, Q. The right-most bit in the register is Q(4), and the left-most bit is Q(1); shifting is performed in the right-to-left direction. The architecture declares the signal *Sreg*,*which* is used to describe the shift operation. All assignments to *Sreg* are synchronized to the clock edge by the IF condition; hence *Sreg* represents the outputs of flip-flops. The statement in line 13 specifies that Sreg(4) is assigned the value of w. As we explained previously, this assignment does not take effect immediately but is scheduled to occur at the end of the process. In line 14 the current value of *Sreg*(4), before it is shifted as a result of line 13, is assigned to Sreg(3). Lines 15 and 16 complete the shift operation. They assign the current values of Sreg(3) and *Sreg(2)*, before they are changed as a result of lines 14 and 15, to *Sreg*(2) and *Sreg*(1), respectively. Finally, *Sreg* is assigned to the Q outputs.

The key point that has to be appreciated in the code in Figure A.38 is that the assignment statements in lines 13 to 16 do not take effect until the end of the process. Hence all

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( w, Clock  : IN     STD_LOGIC ;
5              Q         : OUT  STD_LOGIC_VECTOR(1 TO 4) ) ;
6   END shift4 ;

7   ARCHITECTURE Behavior OF shift4 IS
8       SIGNAL Sreg : STD_LOGIC_VECTOR(1 TO 4) ;
9   BEGIN
10      PROCESS ( Clock )
11      BEGIN
12          IF Clock'EVENT AND Clock = '1' THEN
13              Sreg(4) <= w ;
14              Sreg(3) <= Sreg(4) ;
15              Sreg(2) <= Sreg(3) ;
16              Sreg(1) <= Sreg(2) ;
17          END IF ;
18      END PROCESS ;
19      Q <= Sreg ;
20  END Behavior ;
```

Figure A.38 Code for a four-bit shift register.

flip-flops change their values at the same time, as required in the shift register. We could write the statements in lines 13 to 16 in any order without changing the meaning of the code.

In Section A.9.7 we introduced variables and showed how they differ from signals. As another example of the semantics involved using variables, Figure A.39 gives the code from Figure A.38 but with *Sreg* declared as a variable, instead of as a signal. The statement in line 13 assigns the value of *w* to *Sreg* (4). Since *Sreg* is a variable, the assignment takes effect immediately. In line 14 the value of *Sreg* (4), which has already been changed to w, is assigned to *Sreg* (3). Hence line 14 results in *Sreg* (3) = w. Similarly, lines 15 and 16 set *Sreg* (2) and *Sreg* (1) to the value of w. The code does not describe the desired shift register, but rather loads all flip-flops with the value on the input w.

For the code in Figure A.39 to correctly describe a shift register, the ordering of lines 13 to 16 has to be reversed. Then the first assignment sets

Sreg (1) to the value of *Sreg* (2), the second sets *Sreg* (2) to the value of *Sreg* (3), and so on. Each successive assignment is not affected by the one that precedes it; hence the semantics of using variables does not cause a problem. As we said in Section A.9.7, it can be confusing to use both signals and variables at the same time because they imply different semantics.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( w, Clock  : IN     STD_LOGIC ;
5              Q         : OUT  STD_LOGIC_VECTOR(1 TO 4) ) ;
6   END shift4 ;

7   ARCHITECTURE Behavior OF shift4 IS
8   BEGIN
9       PROCESS ( Clock )
10          VARIABLE Sreg : STD_LOGIC_VECTOR(1 TO 4) ;
11      BEGIN
12          IF Clock'EVENT AND Clock = '1' THEN
13              Sreg(4) := w ;
14              Sreg(3) := Sreg(4) ;
15              Sreg(2) := Sreg(3) ;
16              Sreg(1) := Sreg(2) ;
17          END IF ;
18          Q <= Sreg ;
19      END PROCESS ;
20  END Behavior ;
```

Figure A.39 The code from Figure A.38, using a variable.

A.10.8 COUNTERS

Figure A.40 shows the code for a four-bit counter with an asynchronous reset input. The counter also has an enable input. On the positive clock edge, if the enable E is 1, the count is incremented. If $E = 0$, the counter holds its current value.

A.10.9 USING SUBCIRCUITS WITH GENERIC PARAMETERS

We have shown several examples of VHDL entities that include generic parameters. When these subcircuits are used as components in other code,

the generic parameters can be set to whatever values are needed. To give an example of component instantiation using generics,

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY count4 IS
    PORT ( Resetn     : IN     STD_LOGIC ;
           E, Clock   : IN     STD_LOGIC ;
           Q          : OUT  STD_LOGIC_VECTOR (3 DOWNTO 0) ) ;
END count4 ;

ARCHITECTURE Behavior OF count4 IS
    SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Count <= "0000" ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF E = '1' THEN
                Count <= Count + 1 ;
            END IF ;
        END IF ;
    END PROCESS ;
    Q <= Count ;
END Behavior ;
```

Figure A.40 An example of a counter.

consider the circuit shown in Figure A.41. The circuit adds the binary number represented by the fc-bit input *X* to itself a number of times. Such a circuit is often called an *accumulator.* To store the result of each addition operation, the circuit includes a fc-bit register. The register has an asynchronous reset input, *Resetn.* It also has an enable input, *E*, which is controlled by a four-bit counter. The counter has an asynchronous clear input and a count enable input. The circuit operates by first clearing all bits in the register and counter to 0. Then in each clock cycle, the counter is incremented, and the sum outputs from the adder are stored in the register. When the counter reaches the value 1111, the enable inputs on both the register and counter are set to 0 by the NAND gate. Hence the circuit remains in this state until it is reset again. The final value stored in the register is equal to 15X.

We can represent the accumulator circuit using several subcircuits described in this appendix: *addern* (Figure A.15), *NANDn* (Figure A.28), *regne* (Figure A.37), and *count4* (Figure A.40). We placed the component declaration statements for all of these subcircuits in one package, named *components*, which is shown in Figure A.42.

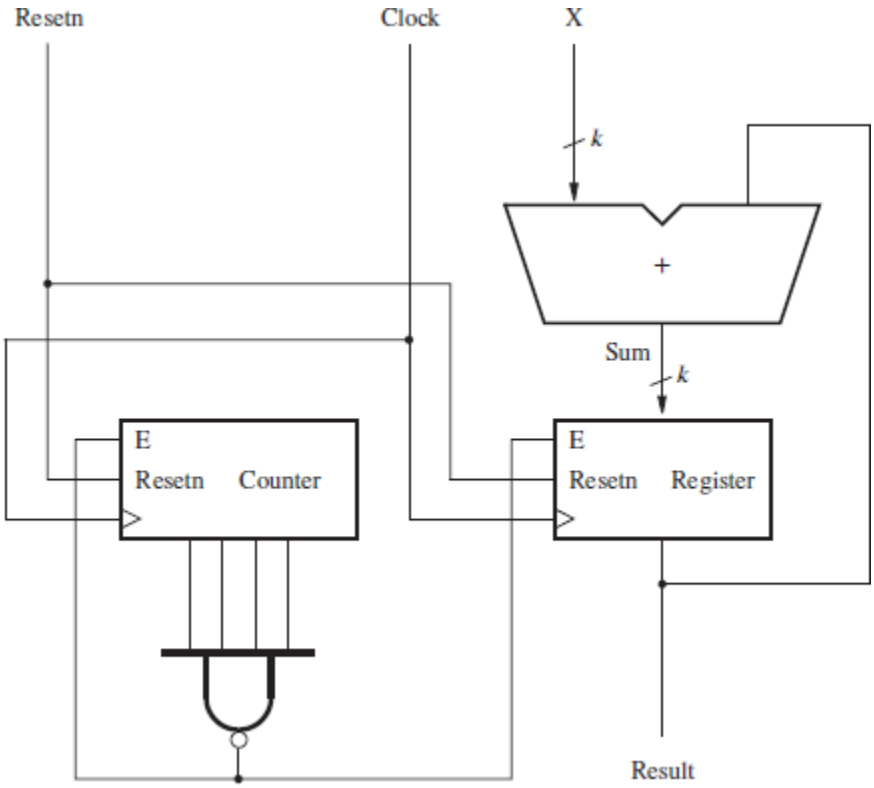


Figure A.41 The accumulator circuit.

Complete code for the accumulator is given in Figure A.43. It uses the generic parameter *k* to represent the number of bits in the input X. Using this parameter in the code makes it easy to change the bit-width at a later time if desired. The architecture defines the signal *Sum* to represent the outputs of the adder; for simplicity, we ignore the possibility of arithmetic overflow and assume that the sum can be represented using *k* bits. The four-bit signal *C* represents the outputs from the counter. The *Stop* signal is connected to the enable inputs on the register and counter.

The statement labeled *adder* instantiates the *addern* subcircuit. The GENERIC MAP keywords are used to specify the value of the adder's generic parameter, n. The syntax (n => k) sets the number of bits in the adder to k. We do not need the carry-in port on the adder, but a signal must

be connected to it. The signal *Zero* Wt,which is setto'0' in the code, is used as a placeholder for the carry-in port (some versions of VHDL compilers do not permit a constant value, such as '1', to be associated directly with a port; hence we define a signal for this purpose). One of the k-bit data inputs to the adder is the primary input X, and the other input to the adder is the Q output of the register, which is named *Result.* The sum output from the adder is named Sum, and the carry-out, which is not used in the circuit, is named *Cout.*

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE components IS

    COMPONENT addern - - n-bit adder
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( Cin    : IN    STD_LOGIC ;
               X, Y   : IN    STD_LOGIC_VECTOR(n–1 DOWNTO 0) ;
               S      : OUT   STD_LOGIC_VECTOR(n–1 DOWNTO 0) ;
               Cout   : OUT   STD_LOGIC ) ;
    END COMPONENT ;

    COMPONENT regne - - n-bit register with enable
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( D          : IN    STD_LOGIC_VECTOR(n–1 DOWNTO 0) ;
               Resetn     : IN    STD_LOGIC ;
               E, Clock   : IN    STD_LOGIC ;
               Q          : OUT   STD_LOGIC_VECTOR(n–1 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT count4 - - 4-bit counter with enable
        PORT ( Resetn     : IN    STD_LOGIC ;
               E, Clock   : IN    STD_LOGIC ;
               Q          : OUT   STD_LOGIC_VECTOR (3 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT NANDn - - n-bit AND gate
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( X   : IN    STD_LOGIC_VECTOR(1 TO n) ;
               f   : OUT   STD_LOGIC ) ;
    END COMPONENT ;

END components ;
```

Figure A.42 Component declarations for the accumulator circuit.

The *regne* subcircuit is instantiated in the statement labeled *reg*. GENERIC MAP is used to set the number of bits in the register to fc.The fc-bit register input is provided by the *Sum* output from the adder. The register's output is named *Result;* this signal represents the output of the accumulator circuit. It has the mode BUFFER in the entity declaration. This is required in the VHDL syntax for the signal to be connected to a port on an instantiated component.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.components.all ;

ENTITY accum IS
    GENERIC ( k : INTEGER := 8 ) ;
    PORT ( Resetn, Clock  : IN      STD_LOGIC ;
           X              : IN      STD_LOGIC_VECTOR(k–1 DOWNTO 0) ;
           Result         : BUFFER  STD_LOGIC_VECTOR(k–1 DOWNTO 0) ) ;
END accum ;

ARCHITECTURE Structure OF accum IS
    SIGNAL Sum : STD_LOGIC_VECTOR(k–1 DOWNTO 0) ;
    SIGNAL C : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
    SIGNAL Zero_bit, Cout, Stop : STD_LOGIC ;
BEGIN
    Zero_bit <= '0' ;
    adder: addern
        GENERIC MAP ( n => k )
        PORT MAP ( Zero_bit, X, Result, Sum, Cout ) ;
    reg: regne
        GENERIC MAP ( n => k )
        PORT MAP ( Sum, Resetn, Stop, Clock, Result ) ;
    Counter: count4
        PORT MAP ( Clock, Resetn, Stop, C ) ;
    NANDgate: NANDn
        PORT MAP ( C, Stop ) ;
END Structure ;
```

Figure A.43 Code for the accumulator circuit.

The *count4* and *NANDn* components are instantiated in the statements labeled *Counter* and *NANDgate*. We do not have to use the GENERIC MAP keyword for *NANDn*, because the default value of its generic parameter is 4, which is the value needed in this application.

A.10.10 A MOORE-TYPE FINITE STATE MACHINE

Figure A.44 shows the state diagram of a simple Moore machine. The code for this machine is shown in Figure A.45. The signal named *y* represents the state of the machine. It is declared with an enumerated type, *State type*, that has the three possible values A, B, and C. When the code is compiled, the VHDL compiler automatically performs a state assignment to select appropriate bit patterns for the three states. The behavior of the machine is defined by the process with the sensitivity list that comprises the reset and clock signals.

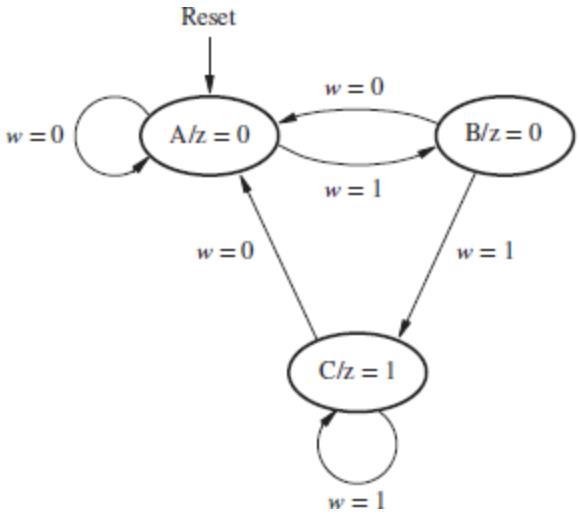


Figure A.44 State diagram of a simple Moore-type FSM.

The VHDL code includes an asynchronous reset input that puts the machine in state A. The state table for the machine is defined using a CASE statement. Each WHEN clause corresponds to a present state of the machine, and the IF statement inside the WHEN clause specifies the next state to be reached after the next positive edge of the clock signal. Since the machine is of the Moore type, the output *z* can be defined as a separate concurrent assignment statement that depends only on the present state of the machine. Alternatively, the appropriate value for *z* could have been specified within each WHEN clause of the CASE statement.

An alternative way to describe a Moore-type finite state machine is given in the architecture in Figure A.46. Two signals are used to describe how the machine moves from one state to another state. The signal *y-present* represents the outputs of the state flip-flops, and the signal *y next* represents the inputs to the flip-flops. The code has two processes. The top process describes a combinational circuit. It uses a CASE statement to specify the

values that *y next* should have for each value of *y-present.* The other process represents a sequential circuit, which specifies that *y-present* is assigned the value of *y next* on the positive clock edge. The process also specifies that *y-present* should take the value *A* when *Resetn* is 0, which provides the asynchronous reset.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY moore IS
    PORT ( Clock   : IN   STD_LOGIC ;
           w       : IN   STD_LOGIC ;
           Resetn  : IN   STD_LOGIC ;
           z       : OUT  STD_LOGIC ) ;
END moore ;

ARCHITECTURE Behavior OF moore IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= B ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= C ;
                    END IF ;
                WHEN C =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= C ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y = C ELSE '0' ;
END Behavior ;
```

Figure A.45 An example of a Moore-type finite state machine.

```
ARCHITECTURE Behavior OF moore IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y_present, y_next : State_type ;
BEGIN
    PROCESS ( w, y_present )
    BEGIN
        CASE y_present IS
            WHEN A =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= B ;
                END IF ;
            WHEN B =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
            WHEN C =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
        END CASE ;
    END PROCESS ;

    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            y_present <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            y_present <= y_next ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

Figure A.46 Code equivalent to Figure A.45, using two processes.

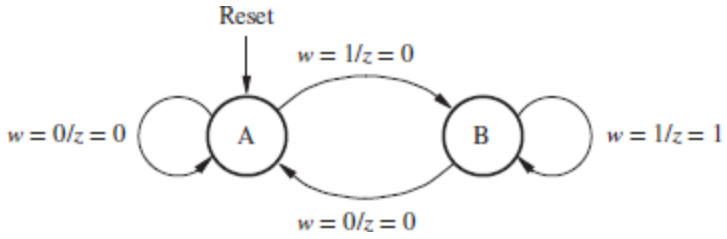


Figure A.47 State diagram of a Mealy-type FSM.

A.10.11 A MEALY-TYPE FINITE STATE MACHINE

A state diagram for a simple Mealy machine is shown in Figure A.47. The corresponding code is given in Figure A.48. The code is the same as in Figure A.45 except that the output *z* is specified using a separate CASE statement. The CASE statement states that when the FSM is in state *A*, *z* should be 0, but when in state B, *z* should take the value of w.ThisCASE statement properly describes the logic needed for *z*. However, it is not obvious why we have used a second CASE statement in the code, rather than specify the value of *z* inside the CASE statement that defines the state table for the machine. This approach would not work properly because the CASE statement for the state table is nested inside the IF statement that waits for a clock edge to occur. Hence if we placed the code for *z* inside this CASE statement, then the value of *z* could change only as a result of a clock edge. This does not meet the requirements of the Mealy-type FSM, because the value of *z* depends not only on the state of the machine but also on the value of the input w.

A.11 COMMON ERRORS IN VHDL CODE

This section lists some common errors that our students have made when writing VHDL code.

ENTITY and ARCHITECTURE Names

The name used in an ENTITY declaration and the corresponding ARCHITECTURE must be identical. The code

```
ENTITY adder IS
⋮
END adder ;
ARCHITECTURE Structure OF adder4 IS
⋮
END Structure ;
```

is erroneous because the ENTITY declaration uses the name *adder*, whereas the architecture uses the name *adder4*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mealy IS
    PORT ( Clock, Resetn  : IN   STD_LOGIC ;
           w              : IN   STD_LOGIC ;
           z              : OUT  STD_LOGIC ) ;
END mealy ;

ARCHITECTURE Behavior OF mealy IS
    TYPE State_type IS (A, B) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    PROCESS ( y, w )
    BEGIN
        CASE y IS
            WHEN A =>
                z <= '0' ;
            WHEN B =>
                z <= w ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure A.48 An example of a Mealy-type machine.

Missing Semicolon

Every VHDL statement must end with a semicolon.

Use of Quotes

Single quotes are used for single-bit data, double quotes for multibit data, and no quotes are used for integer data. Examples are given in Section A.2.

Combinational versus Sequential Statements

Combinational statements include simple signal assignments, selected signal assignments, conditional signal assignments, and generate statements. Simple signal assignments can be used either outside or inside a PROCESS statement. The other types of combinational statements can be used only outside a PROCESS statement.

Sequential statements include IF, CASE, and LOOP statements. Each of these types of statements can be used only inside a process statement.

Component Instantiation

The following statement contains two potential errors

```
control: shiftr GENERIC MAP ( K => 3 ) ;
                PORT MAP ( '1' , Clock, w, Q ) ;
```

There should be no semicolon at the end of the first line, because the two lines represent a single VHDL statement. Also, some VHDL compilers do not support the use of a constant value ('1') with a port on a component. The following code shows how the two errors can be fixed

```
SIGNAL High ;
:
High <= '1' ;
control: shiftr GENERIC MAP ( K => 3 )
                PORT MAP ( High, Clock, w, Q ) ;
```

Label, Signal, and Variable Names

It is illegal to use any VHDL keyword as a label, signal, or variable name. For example, it is illegal to name a signal *In* or *Out.* Also, it is illegal to use the same name multiple times for any label, signal, or variable in a given VHDL design. A common error is to use the same name for a signal and a variable used as the index in a generate or loop statement. For instance, if the code uses the generate statement

```
Generate_label:
FOR i IN 0 TO 3 GENERATE
    bit: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
END GENERATE ;
```

then it is illegal to define a signal named *i* (or I, because VHDL does not distinguish between lower and uppercase letters).

Implied Memory

As shown in Section A.10, implied memory is used to describe storage elements. Care must be taken to avoid unintentional implied memory. The code

```
IF LA = '1' THEN
    EA <= '1' ;
END IF ;
```

results in implied memory for the *EA* signal. If this is not desired, then the code can be fixed by writing

```
IF LA = '1' THEN
    EA <= '1' ;
ELSE
    EA <= '0' ;
END IF ;
```

Implied memory also applies to CASE statements. The statement

```
CASE y IS
    WHEN S1 =>
        EA <= '1' ;
    WHEN S2 =>
        EB <= '1' ;
END CASE ;
```

does not specify the value of the *EA* signal when *y* is not equal to 51, and it does not specify the value of *EB* when *y* is not equal to 52. To avoid having implied memory for both *EA* and EB, these signals should be assigned default values, as in the code

```
EA <= '0' ; EB <= '0' ;
CASE y IS
    WHEN S1 =>
        EA <= '1' ;
    WHEN S2 =>
        EB <= '1' ;
END CASE ;
```

A.12 Concluding Remarks

This appendix describes all the important VHDL constructs that are useful for the synthesis of logic circuits. As mentioned earlier, we do not discuss any features of VHDL that are useful only for simulation of circuits, or for other purposes. A reader who wishes to learn more about using VHDL can refer to specialized books [1-8].

References

1. Institute of Electrical and Electronics Engineers, "1076-1993 IEEE Standard VHDL Language Reference Manual," 1993.
2. D. L. Perry, *VHDL*, 3rd ed. (McGraw-Hill: New York, 1998).
3. Z. Navabi, *VHDL-Analysis and Modeling of Digital Systems*, 2nd ed. (McGraw-Hill: New York, 1998).
4. J. Bhasker, *A VHDL Primer*, 3rd ed. (Prentice-Hall: Englewood Cliffs, NJ, 1998).
5. K. Skahill, *VHDL for Programmable Logic* (Addison-Wesley: Menlo Park, CA, 1996).
6. A. Dewey, *Analysis and Design ofDigital Systems with VHDL* (PWS Publishing Co.: Boston, MA, 1997).
7. S. Yalamanchili, *VHDL Starter's Guide* (Prentice-Hall: Upper Saddle River, NJ, 1998).

8. P. Ashenden, *The Designer's Guide to VHDL*, 2nd ed. (Morgan Kaufmann: San Francisco, CA, 2001).

appendix

B

IMPLEMENTATION TECHNOLOGY

This appendix provides a detailed discussion of integrated circuit technology. We show how transistors operate as simple switches, and how logic gates are built using transistors. We also describe the structure of integrated circuit chips that are used to implement logic circuits, including programmable logic devices, gate arrays, and memory chips. Finally, we study some fundamental properties of electronic circuits, including current flow and power dissipation.

Let us first consider how logic variables can be physically represented as signals in electronic circuits. Our discussion will be restricted to binary variables, which can take on only the values 0 and 1. In a circuit these values can be represented either as levels of voltage or current. Both alternatives are used in different technologies. We will focus on the simplest and most popular representation, using voltage levels.

The most obvious way of representing two logic values as voltage levels is to define a *threshold* voltage; any voltage below the threshold represents one logic value, and voltages above the threshold correspond to the other logic value. It is an arbitrary choice as to which logic value is associated with the low and high voltage levels. Usually, logic 0 is represented by the low voltage levels and logic 1 by the high voltages. This is known as a *positive logic* system. The opposite choice, in which the low voltage levels are used to represent logic 1 and the higher voltages are used for logic 0 is known as a

negative logic system. In this book we use only the positive logic system, but negative logic is discussed briefly in Section B.4.

Using the positive logic system, the logic values 0 and 1 are referred to simply as "low" and "high." To implement the threshold-voltage concept, a range of low and high voltage levels is defined, as shown in Figure B.1. The figure gives the minimum voltage, called $***V_{SS}$, and the maximum voltage, called V_{DD}, that can exist in the circuit. We will assume that V_{SS} is 0 volts, corresponding to electrical ground, denoted *Gnd*. The voltage V_{DD} represents the power supply voltage. In some technologies V_{DD} may be as large as 5 volts, while in others it could be as small as 1 volt, or even less. In this appendix we will mostly use the value V_{DD}=5 V. Figure B.1 indicates that voltages in the range *Gnd* to $V_{0,max}$ represent logic value 0. The name $V_{0,max}$ means the maximum voltage level that a logic circuit must recognize as low. Similarly, the range from $V_{1,min}$ to V_{DD} corresponds to logic value 1, and $V_{1,min}$ is the minimum voltage level that a logic circuit must interpret as high. The exact levels of $V_{0,max}$ and $V_{1,min}$ depend on the particular technology used; a typical example might set $V_{0,max}$ to 40 percent of V_{DD} and $V_{1,min}$ to 60 percent of V_{DD}. The range of voltages between $V_{0,max}$ and $V_{1,min}$ is undefined. Logic signals do not normally assume voltages in this range except in transition from one logic value to the other. We will discuss the voltage levels used in logic circuits in more depth in Section B.8.3.

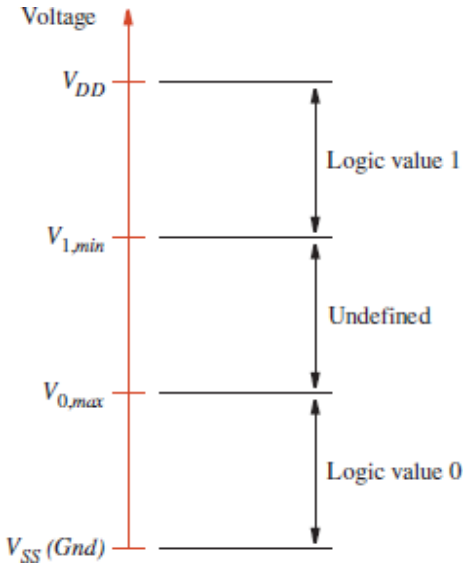
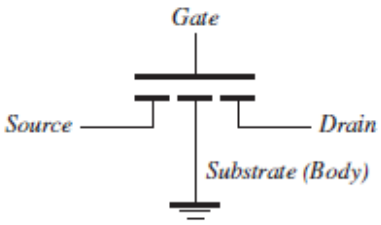


Figure B.1 Representation of logic values by voltage levels.

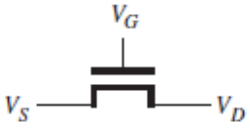
B.1 TRANSISTOR SWITCHES

Logic circuits are built with transistors. A full treatment of transistor behavior is beyond the scope of this text; it can be found in electronics textbooks, such as [1] and [2]. For the purpose of understanding how logic circuits are built, we can assume that a transistor operates as a simple switch. Figure B.2*a* shows a switch controlled by a logic signal, *x*. When *x* is low, the switch is open, and when *x* is high, the switch is closed. The most popular type of transistor for implementing a simple switch is the *metal oxide semiconductor field-effect transistor (MOSFET)*. There are two different types of MOSFETs, known as *n-channel*, abbreviated *NMOS*, and *p-channel*, denoted *PMOS*.

(a) A simple switch controlled by the input *x*



(b) NMOS transistor



(c) Simplified symbol for an NMOS transistor

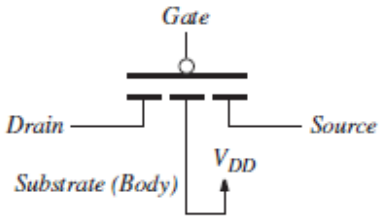
Figure B.2 NMOS transistor as a switch.

Figure B.2*b* gives a graphical symbol for an NMOS transistor. It has four electrical terminals, called the *source*, *drain*, *gate*, and *substrate*. In logic circuits the substrate (also called *body*) terminal is connected to *Gnd*. We will use the simplified graphical symbol in Figure B.2*c*, which omits the substrate node. There is no physical difference between the source and drain terminals. They are distinguished in practice by the voltage levels applied to the

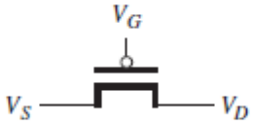
transistor; by convention, the terminal with the lower voltage level is deemed to be the source.

A detailed explanation of how the transistor operates will be presented in Section B.8.1. For now it is sufficient to know that it is controlled by the voltage V_G at the gate terminal. If V_G is low, then there is no connection between the source and drain, and we say that the transistor is *turned off*. If V_G is high, then the transistor is *turned on* and acts as a closed switch that connects the source and drain terminals. In Section B.8.2 we show how to calculate the resistance between the source and drain terminals when the transistor is turned on, but for now assume that the resistance is 0 Ω.

(a) A switch with the opposite behavior of Figure 3.2*a*



(b) PMOS transistor



(c) Simplified symbol for an PMOS transistor

Figure B.3 PMOS transistor as a switch.

PMOS transistors have the opposite behavior of NMOS transistors. The former are used to realize the type of switch illustrated in Figure B.3*a*, where the switch is open when the control input x is high and closed when x is low. A symbol is shown in Figure B.3*b*. In logic circuits the substrate of the PMOS transistor is connected to V_{DD}, leading to the simplified symbol in Figure B.3*c*. If V_G is high, then the PMOS transistor is turned off and acts like an open switch. When V_G is low, the transistor is turned on and acts as a closed

switch that connects the source and drain. In the PMOS transistor the source is the node with the higher voltage.

Figure B.4 summarizes the typical use of NMOS and PMOS transistors in logic circuits. An NMOS transistor is turned on when its gate terminal is high, while a PMOS transistor is turned on when its gate is low. When the NMOS transistor is turned on, its drain is *pulled down* to *Gnd*, and when the PMOS transistor is turned on, its drain is *pulled up* to V_{DD}. Because of the way the transistors operate, an NMOS transistor cannot be used to pull its drain terminal completely up to V_{DD}. Similarly, a PMOS transistor cannot be used to pull its drain terminal completely down to *Gnd*. We discuss the operation of MOSFETs in considerable detail in Section B.8.

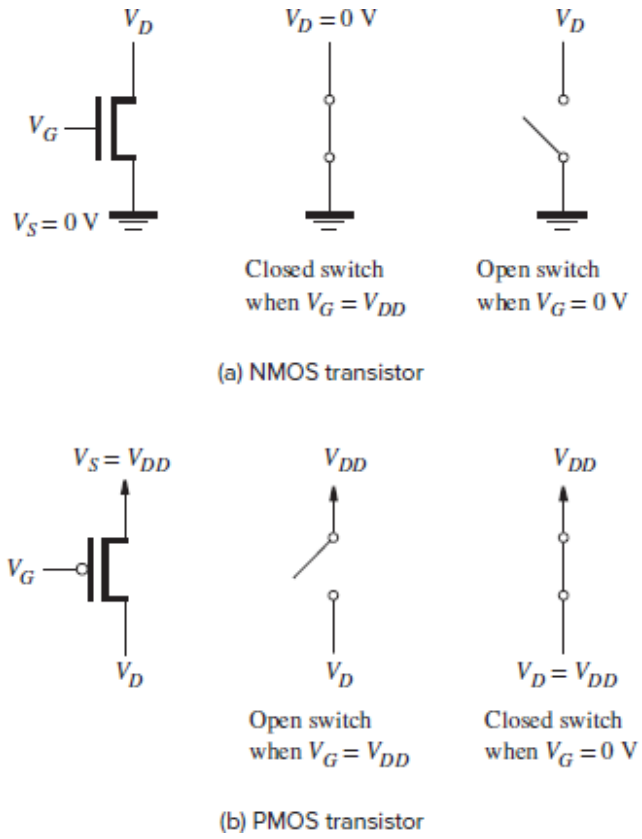


Figure B.4 NMOS and PMOS transistors in logic circuits.

B.2 NMOS Logic Gates

The first schemes for building logic gates with MOSFETs became popular in the 1970s and relied on either PMOS or NMOS transistors, but not both. Since the early 1980s, a combination of both NMOS and PMOS transistors has been used. We will first describe how logic circuits can be built using NMOS transistors because these circuits are easier to understand. Such circuits are known as NMOS circuits. Then we will show how NMOS and PMOS transistors are combined in the presently popular technology known as *complementary MOS*, or *CMOS*.

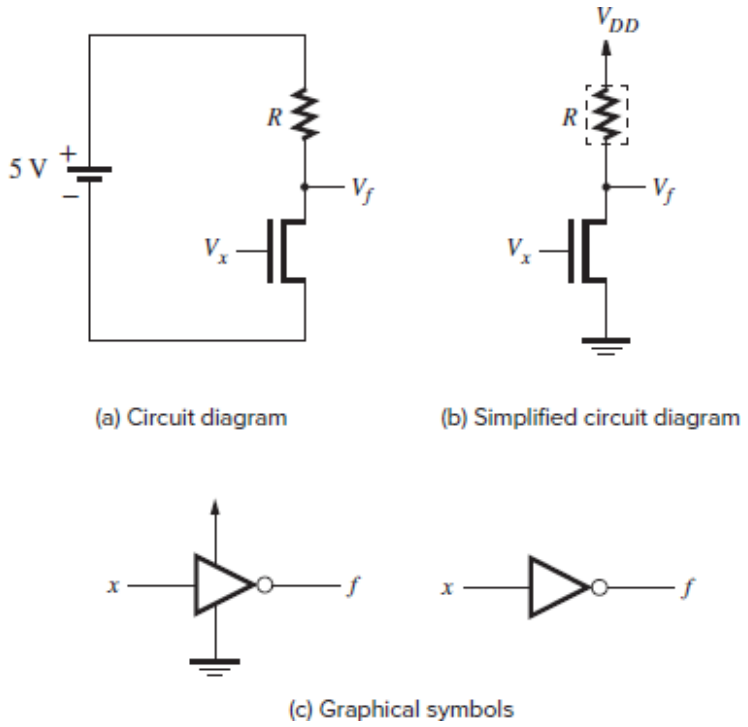


Figure B.5 A NOT gate built using NMOS technology.

In the circuit in Figure B.5*a*, when V_x = 0 V the NMOS transistor is turned off. No current flows through the resistor R, and V_f = 5 V. On the other hand, when V_x = 5 V, the transistor is turned on and pulls V_f to a low voltage level. The exact voltage level of V_f in this case depends on the amount of current that flows through the resistor and transistor. Typically, V_f is about 0.2 V (see Section B.8.3). If V_f is viewed as a function of V_x, then the circuit is an NMOS implementation of a NOT gate. In logic terms this circuit implements the function $f = \bar{x}$. Figure B.5*b* gives a simplified circuit diagram in which the connection to the positive terminal on the power supply is indicated by an arrow labeled V_{DD} and the connection to the negative power-supply terminal

is indicated by the *Gnd* symbol. We will use this simplified style of circuit diagram for the figures that follow.

The purpose of the resistor in the NOT gate circuit is to limit the amount of current that flows when V_x = 5 V. Rather than using a resistor for this purpose, a transistor is normally used. We will discuss this issue in more detail in Section B.8.3. In subsequent diagrams a dashed box is drawn around the resistor *R* as a reminder that it is implemented using a transistor.

Figure B.5*c* presents the graphical symbols for a NOT gate. The symbol on the left shows the input, output, power, and ground terminals, and the symbol on the right is simplified to show only the input and output terminals. In practice only the simplified symbol is used. Another name often used for the NOT gate is *inverter*. We use both names interchangeably in this book.

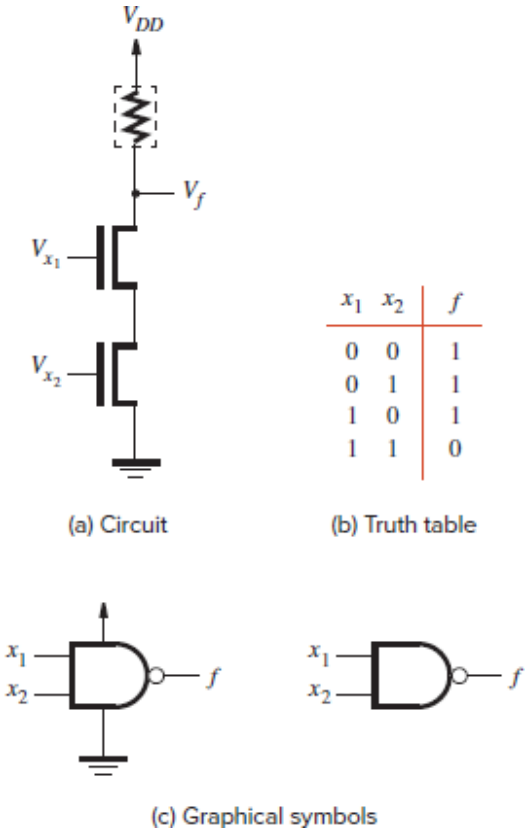


Figure B.6 NMOS realization of a NAND gate.

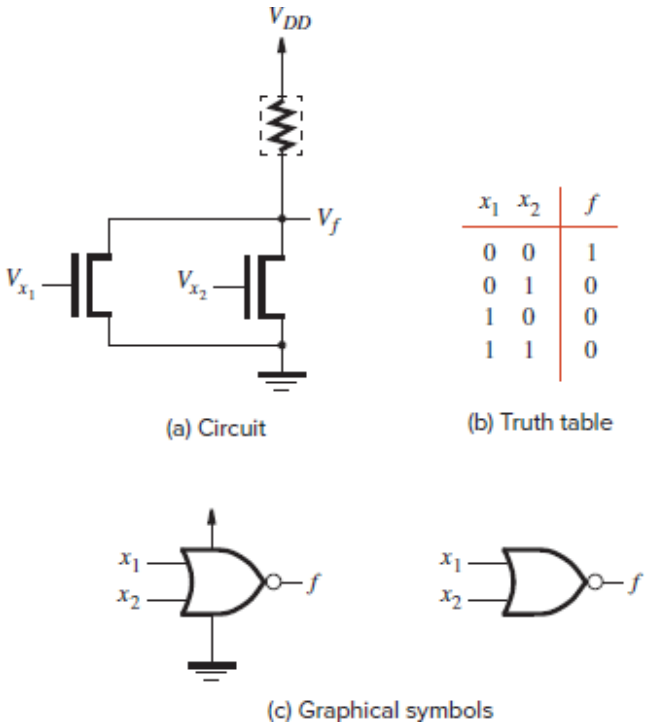


Figure B.7 NMOS realization of a NOR gate.

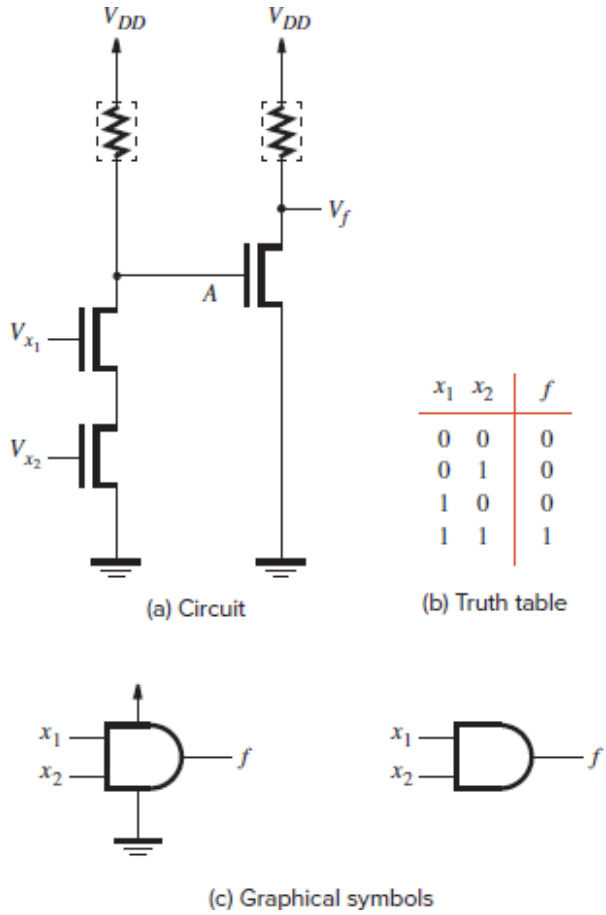


Figure B.8 NMOS realization of an AND gate.

In Section 2.1 we saw that a series connection of switches corresponds to the logic AND function, while a parallel connection represents the OR function. Using NMOS transistors, we can implement the series connection as depicted in Figure B.6*a*. If $V_{x_1} = V_{x_2} = 5$ V, both transistors will be on and V_f will be close to 0 V. But if either V_{x_1} or V_{x_2} is 0, then no current will flow through the series-connected transistors and V_f will be pulled up to 5 V. The resulting truth table for *f*, provided in terms of logic values, is given in Figure B.6*b*. The circuit realizes a NAND gate. Its graphical symbols are shown in part (*c*) of the figure.

The parallel connection of NMOS transistors is given in Figure B.7*a*. Here, if either $V_{x_1} = 5$ V or $V_{x_2} = 5$ V, then V_f will be close to 0 V. Only if both V_{x_1} and V_{x_2} are 0 will V_f be pulled up to 5 V. A corresponding truth table is given in Figure B.7*b*, which shows that the circuit realizes the NOR function. The graphical symbols for the NOR gate appear in part (*c*) of the figure.

Figure B.8 indicates how an AND gate is built in NMOS technology by following a NAND gate with an inverter. Node *A* realizes the NAND of inputs x_1 and x_2, and *f* represents the AND function. In a similar fashion an OR gate is realized as a NOR gate followed by an inverter, as depicted in Figure B.9.

B.3 CMOS Logic Gates

So far we have considered the implementation of logic gates using NMOS transistors. For each of the circuits that has been presented, it is possible to derive an equivalent circuit that uses PMOS transistors. However, it is more interesting to consider how both NMOS and PMOS transistors can be used together. The most popular such approach is known as CMOS technology. We will see in Section B.8 that CMOS technology offers some attractive practical advantages in comparison to NMOS technology.

In NMOS circuits the logic functions are realized by arrangements of NMOS transistors, combined with a pull-up device that acts as a resistor. We will refer to the part of the circuit that involves NMOS transistors as the *pull-down network (PDN).* Then the structure of the circuits in Figures B.5 through B.9 can be characterized by the block diagram in Figure B.10. The concept of CMOS circuits is based on replacing the pull-up device with a *pull-up network (PUN)* that is built using PMOS transistors, such that the functions realized by the PDN and PUN networks are complements of each other. Then a logic circuit, such as a typical logic gate, is implemented as indicated in Figure B.11. For any given valuation of the input signals, either the PDN pulls V_f down to *Gnd* or the PUN pulls V_f up to V_{DD}. The PDN and the PUN have equal numbers of transistors, which are arranged so that the two networks are *duals* of one another. Wherever the PDN has NMOS transistors in series, the PUN has PMOS transistors in parallel, and vice versa.

The simplest example of a CMOS circuit, a NOT gate, is shown in Figure B.12. When $V_x = 0$ V, transistor T_2 is off and transistor T_1 is on. This makes $V_f = 5$ V, and since T_2 is off, no current flows through the transistors. When $V_x = 5$ V, T_2 is on and T_1 is off. Thus $V_f = 0$ V, and no current flows because T_1 is off.

A key point is that no current flows in a CMOS inverter when the input is either low or high. This is true for all CMOS circuits; no current flows, and hence no power is dissipated under steady state conditions. This property has

led to CMOS becoming the most popular technology in use today for building logic circuits. We will discuss current flow and power dissipation in detail in Section B.8.6.

Figure B.13 provides a circuit diagram of a CMOS NAND gate. It is similar to the NMOS circuit presented in Figure B.6 except that the pull-up device has been replaced by the PUN with two PMOS transistors connected in parallel. The truth table in the figure specifies the state of each of the four transistors for each logic valuation of inputs x_1 and x_2. The reader can verify that the circuit properly implements the NAND function. Under static conditions no path exists for current flow from V_{DD} to *Gnd*.

The circuit in Figure B.13 can be derived from the logic expression that defines the NAND operation, $f = \overline{x_1 x_2}$. This expression specifies the conditions for which f=1; hence it defines the PUN. Since the PUN consists of PMOS transistors, which are turned on when their control (gate) inputs are set to 0, an input variable x_i turns on a transistor if x_i = 0. From DeMorgan's law, we have

$$f = \overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$$

Thus f=1 when *either* input x_1 or x_2 has the value 0, which means that the PUN must have two PMOS transistors connected in parallel. The PDN must implement the complement of f, which is

$$\bar{f} = x_1 x_2$$

Since $\bar{f} = 1$ when *both* x_1 and x_2 are 1, it follows that the PDN must have two NMOS transistors connected in series.

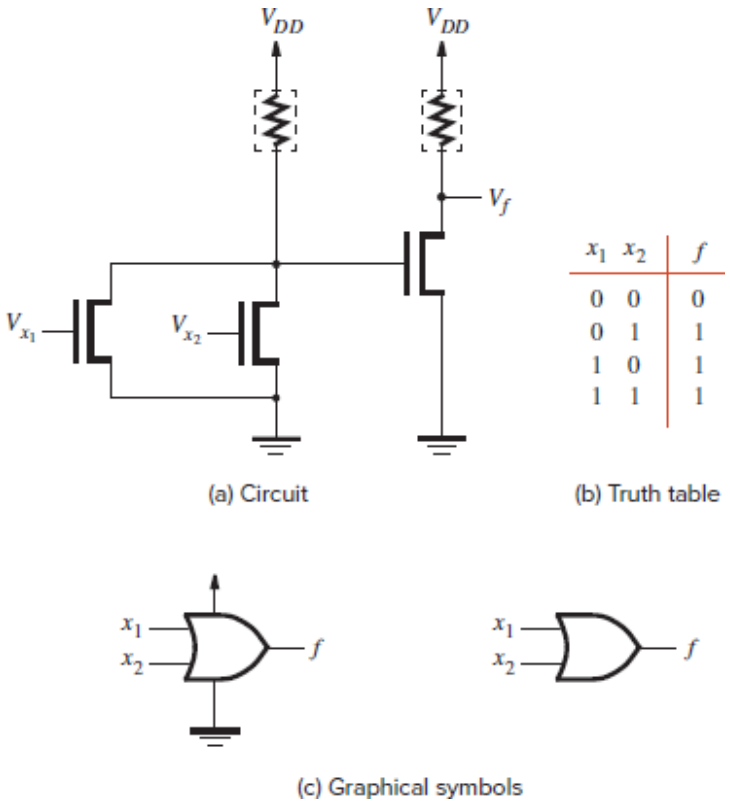


Figure B.9 NMOS realization of an OR gate.

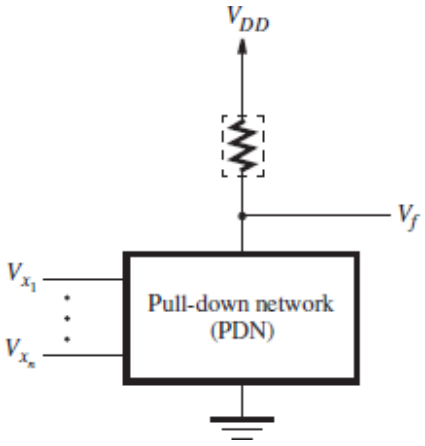


Figure B.10 Structure of an NMOS circuit.

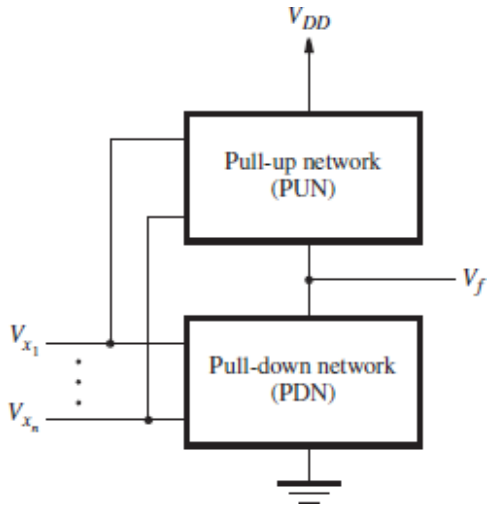
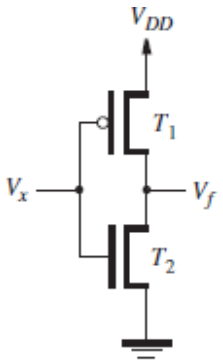


Figure B.11 Structure of a CMOS circuit.

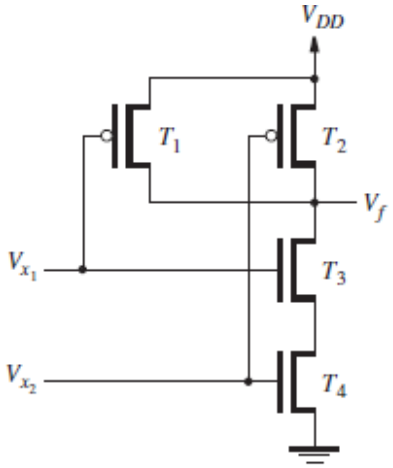


x	T_1	T_2	f
0	on	off	1
1	off	on	0

(a) Circuit

(b) Truth table and transistor states

Figure B.12 CMOS realization of a NOT gate.

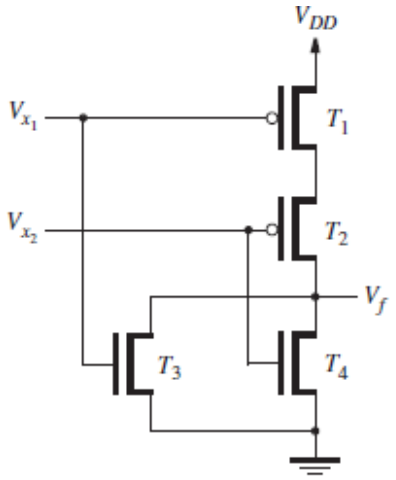


x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

(a) Circuit

(b) Truth table and transistor states

Figure B.13 CMOS realization of a NAND gate.



(a) Circuit

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	0
1	0	off	on	on	off	0
1	1	off	off	on	on	0

(b) Truth table and transistor states

Figure B.14 CMOS realization of a NOR gate.

The circuit for a CMOS NOR gate is derived from the logic expression that defines the NOR operation

$$f = \overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$$

Since f=1 only if both x_1 and x_2 have the value 0, then the PUN consists of two PMOS transistors connected in series. The PDN, which realizes $\bar{f} = x_1 + x_2$, has two NMOS transistors in parallel, leading to the circuit shown in Figure B.14.

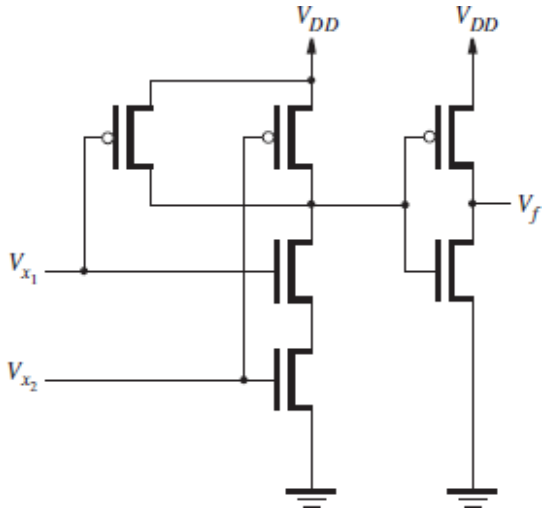


Figure B.15 CMOS realization of an AND gate.

A CMOS AND gate is built by connecting a NAND gate to an inverter, as illustrated in Figure B.15. Similarly, an OR gate is constructed with a NOR gate followed by a NOT gate.

The above procedure for deriving a CMOS circuit can be applied to more general logic functions to create *complex gates*. This process is illustrated in the following two examples.

Example B.1

Consider the function

$$f = \bar{x}_1 + \bar{x}_2\bar{x}_3$$

Since all variables appear in their complemented form, we can directly derive the PUN. It consists of a PMOS transistor controlled by x_1 in parallel with a series combination of PMOS transistors controlled by x_2 and x_3. For the PDN we have

$$\bar{f} = \overline{\bar{x}_1 + \bar{x}_2\bar{x}_3} = x_1(x_2 + x_3)$$

This expression gives the PDN that has an NMOS transistor controlled by x_1 in series with a parallel combination of NMOS transistors controlled by x_2 and x_3. The circuit is shown in Figure B.16.

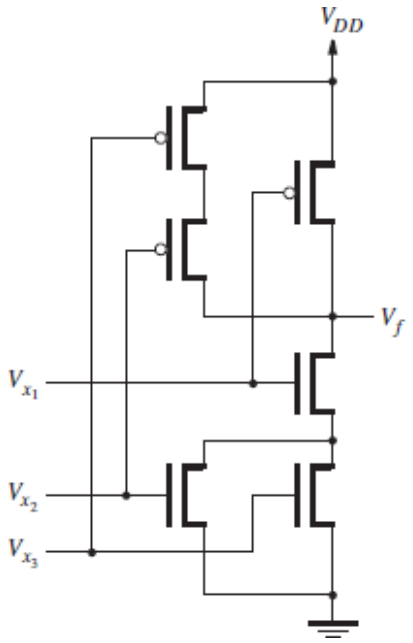


Figure B.16 The circuit for Example B.1.

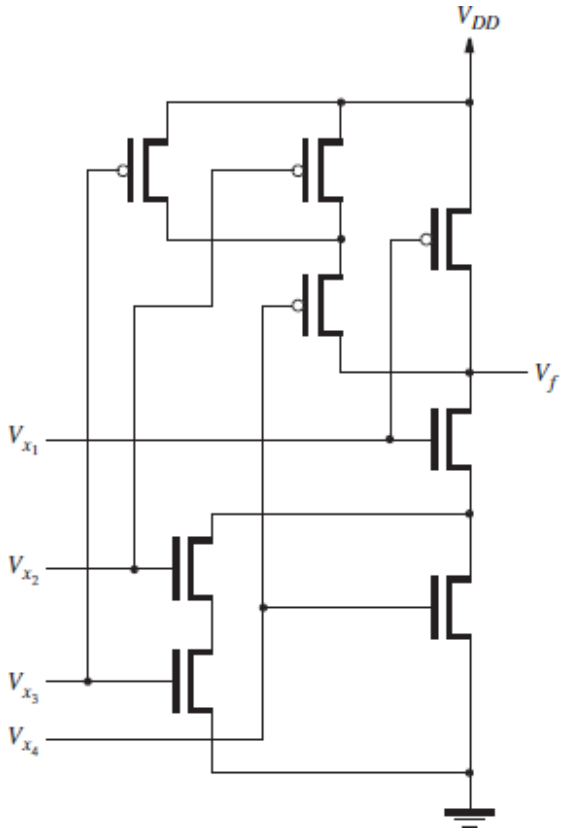


Figure B.17 The circuit for Example B.2.

Example B.2

consider the function

$$f = \bar{x}_1 + (\bar{x}_2 + \bar{x}_3)\bar{x}_4$$

Then

$$\bar{f} = x_1(x_2x_3 + x_4)$$

These expressions lead directly to the circuit in Figure B.17.

The circuits in Figures B.16 and B.17 show that it is possible to implement fairly complex logic functions using combinations of series and parallel connections of transistors (acting as switches), without implementing each series or parallel connection as a complete AND (using the structure introduced in Figure B.15) or OR gate.

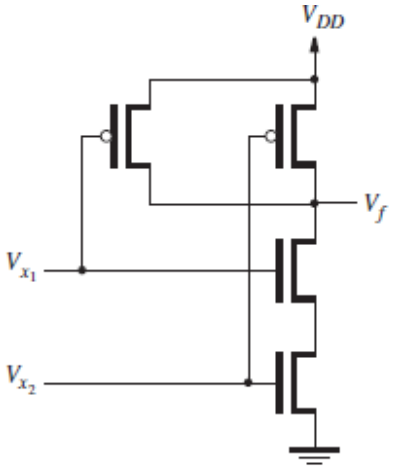
B.3.1 Speed of Logic Gate Circuits

In the preceding sections we have assumed that transistors operate as ideal switches that present no resistance to current flow. Hence, while we have derived circuits that realize the functionality needed in logic gates, we have ignored the important issue of the speed of operation of the circuits. In reality transistor switches have a significant resistance when turned on. Also, transistor circuits include capacitors, which are created as a side effect of the manufacturing process. These factors affect the amount of time required for signal values to propagate through logic gates. We provide a detailed discussion of the speed of logic circuits, as well as a number of other practical issues, in Section B.8.

B.4 Negative Logic System

In the discussion for Figure B.1, we said that logic values are represented as two distinct ranges of voltage levels. We are using the convention that the higher voltage levels represent logic value 1 and the lower voltages represent logic value 0. This convention is known as the positive logic system, and it is

the one used in most practical applications. In this section we briefly consider the negative logic system in which the association between voltage levels and logic values is reversed.



(a) Circuit

V_{x_1}	V_{x_2}	V_f
L	L	H
L	H	H
H	L	H
H	H	L

(b) Voltage levels

Figure B.18 Voltage levels in the circuit in Figure B.13.

x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

(a) Positive logic truth table and gate symbol

x_1	x_2	f
1	1	0
1	0	0
0	1	0
0	0	1

(b) Negative logic truth table and gate symbol

Figure B.19 Interpretation of the circuit in Figure B.18.

Let us reconsider the CMOS circuit in Figure B.13, which is reproduced in Figure B.18*a*. Part (*b*) of the figure gives a truth table for the circuit, but the table shows voltage levels instead of logic values. In this table, *L* refers to the low voltage level in the circuit, which is 0 V, and *H* represents the high voltage level, which is V_{DD}. This is the style of truth table that manufacturers

of integrated circuits often use in data sheets to describe the functionality of the chips. It is entirely up to the user of the chip as to whether *L* and *H* are interpreted in terms of logic values such that *L*=0 and *H*=1, or *L*=1 and *H*=0.

Figure B.19*a* illustrates the positive logic interpretation in which *L*=0 and *H*=1. As we already know from the discussions of Figure B.13, the circuit represents a NAND gate under this interpretation. The opposite interpretation is shown in Figure B.19*b*. Here negative logic is used so that *L*=1 and *H*=0. The truth table specifies that the circuit represents a NOR gate in this case. Note that the truth table rows are listed in the opposite order from what we normally use, to be consistent with the *L* and *H* values in Figure B.18*b*. Figure B.19*b* also gives the logic gate symbol for the NOR gate, which includes small triangles on the gate's terminals to indicate that the negative logic system is used.

As another example, consider again the circuit in Figure B.15. Its truth table, in terms of voltage levels, is given in Figure B.20*a*. Using the positive logic system, this circuit represents an AND gate, as indicated in Figure B.20*b*. But using the negative logic system, the circuit represents an OR gate, as depicted in Figure B.20*c*.

V_{x_1}	V_{x_2}	V_f
L	L	L
L	H	L
H	L	L
H	H	H

(a) Voltage levels

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

(b) Positive logic

x_1	x_2	f
1	1	1
1	0	1
0	1	1
0	0	0

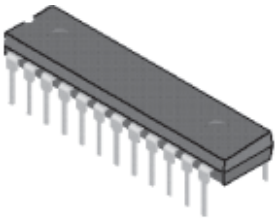
(c) Negative logic

Figure B.20 Interpretation of the circuit in Figure B.15.

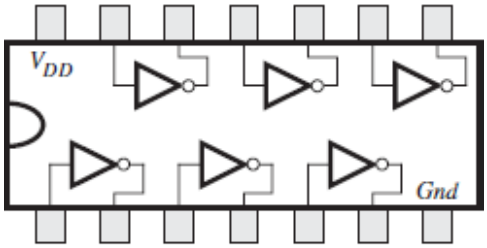
It is possible to use a mixture of positive and negative logic in a single circuit, which is known as a *mixed logic system*. In practice, the positive logic system is used in most applications. We do not consider the negative logic system in the main body of the book.

B.5 Standard Chips

In Chapter 1 we mentioned that several different types of integrated circuit chips are available for implementation of logic circuits. We now discuss the available choices in some detail.



(a) Dual-inline package



(b) Structure of 7404 chip

Figure B.21 A 7400-series chip.

B.5.1 7400-SERIES STANDARD CHIPS

An approach used widely until the mid-1980s was to connect together multiple chips, each containing only a few logic gates. A wide assortment of chips, with different types of logic gates, is available for this purpose. They are known as 7400-series parts because the chip part numbers always begin with the digits 74. An example of a 7400-series part is given in Figure B.21. Part (*a*) of the figure shows a type of package that the chip is provided in, called a *dual-inline package (DIP).* Part (*b*) illustrates the 7404 chip, which comprises six NOT gates. The chip's external connections are called *pins* or *leads*. Two pins are used to connect to V_{DD} and *Gnd*, and other pins provide connections to the NOT gates. Many 7400-series chips exist, and they are described in the data books produced by manufacturers of these chips [3–7]. Diagrams of some of the chips are also included in several textbooks, such as [8–12].

The 7400-series chips are produced in standard forms by a number of integrated circuit manufacturers, using agreed-upon specifications. Competition among various manufacturers works to the designer's advantage because it tends to lower the price of chips and ensures that parts are always readily available. For each specific 7400-series chip, several variants are built with different technologies. For instance, the part called 74LS00 is built with a

technology called transistor-transistor logic (TTL), which is described in [1], whereas the 74HC00 is fabricated using CMOS technology. In general, the most popular chips used today are the CMOS variants.

As an example of how a logic circuit can be implemented using 7400-series chips, consider the function $f = x_1x_2 + \bar{x}_2x_3$. A NOT gate is required to produce $\bar{x}_2$, as well as 2 two-input AND gates and a two-input OR gate. Figure B.22 shows three 7400-series chips that can be used to implement the function. We assume that the three input signals x_1, x_2, and x_3 are produced as outputs by some other circuitry that can be connected by wires to the three chips. Notice that power and ground connections are included for all three chips. This example makes use of only a portion of the gates available in the three chips, hence the remaining gates can be used to realize other functions.

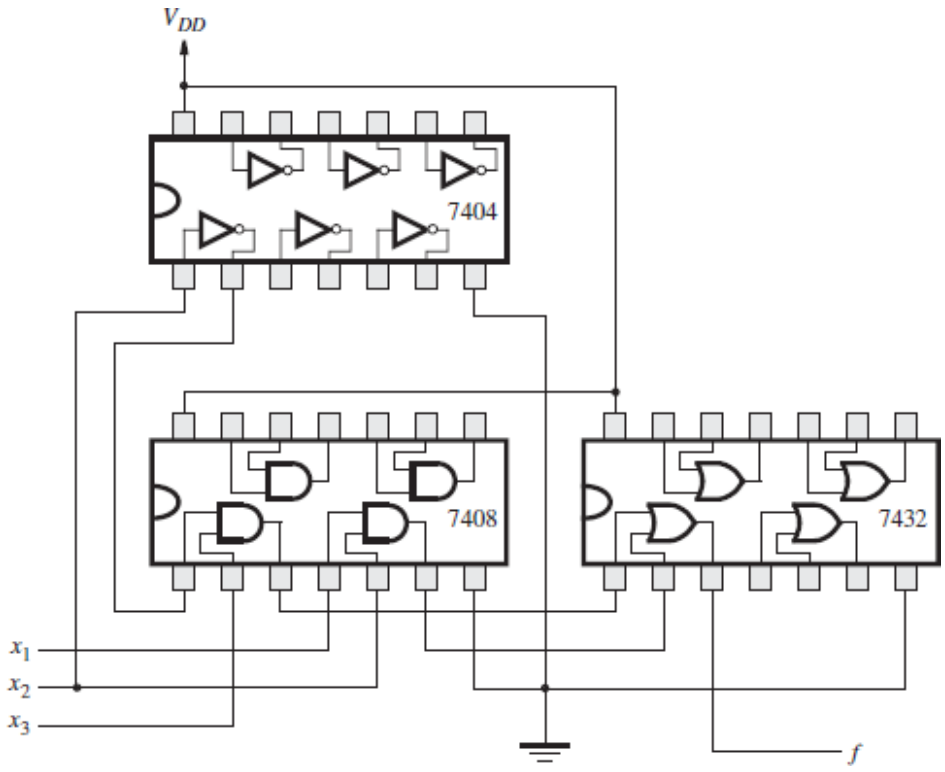


Figure B.22 An implementation of $f = x_1x_2 + \bar{x}_2x_3$.

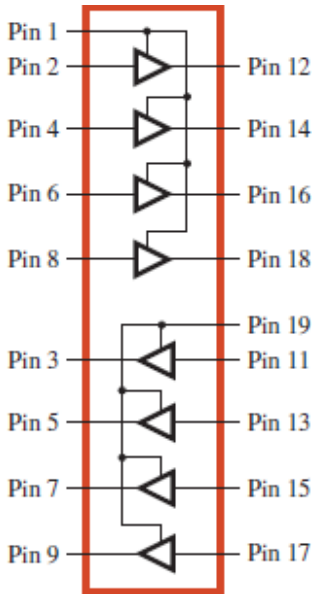


Figure B.23 The 74244 driver chip.

Because of their low logic capacity, the standard chips are seldom used in practice today, with one exception. Many modern products include standard chips that contain drivers. Drivers (also sometimes called *buffers*) are logic gates that are usually used to improve the speed of circuits. An example of a driver chip is depicted in Figure B.23. It is the 74244 chip, which comprises eight *tri-state drivers*. We describe how tri-state drivers work in Section 7.1.1. Rather than showing how the drivers are arranged inside the chip package, as we did for the NOT gates in Figure B.21, we show only the pin numbers of the package pins that are connected to the drivers. The package has 20 pins, and they are numbered with pin 1 in the lower-left corner, and pin 20 in the upper-left corner. The *Gnd* and V_{DD} connections are provided on pins 10 and 20, respectively, similar to the arrangement shown in Figure B.21. Many other driver chips also exist. For example, the 162244 chip has 16 tri-state drivers. It is part of a family of devices that are similar to the 7400-series chips but with twice as many gates in each chip. These chips are available in multiple types of packages, with the most popular being a *small-outline integrated circuit (SOIC)* package. An SOIC package has a similar shape to a DIP, but the SOIC is considerably smaller in physical size.

As integrated circuit technology has improved over time, a system of classifying chips according to their size has evolved. The earliest chips produced, such as the 7400-series, comprise only a few logic gates. The technology used to produce these chips is referred to as *small-scale*

integration (SSI). Chips that include slightly more logic circuitry, typically about 10 to 100 gates, represent *medium-scale integration (MSI)*. Until the mid-1980s chips that were too large to qualify as MSI were classified as *large-scale integration (LSI)*. In recent years the concept of classifying circuits according to their size has become of little practical use. Most integrated circuits today contain millions, or even billions, of transistors. Regardless of their exact size, these large chips are said to be made with *very large scale integration (VLSI)* technology. The trend in digital hardware products is to integrate as much circuitry as possible onto a single chip. Thus most of the chips used today are built with VLSI technology, and the older types of chips are used rarely.

B.6 Programmable Logic Devices

The function provided by each of the 7400-series parts is fixed and cannot be tailored to suit a particular design situation. This fact, coupled with the limitation that each chip contains only a few logic gates, makes these chips inefficient for building large logic circuits. It is possible to manufacture chips that contain relatively large amounts of logic circuitry with a structure that is not fixed. Such chips were first introduced in the 1970s and are called *programmable logic devices (PLDs)*.

A PLD is a general-purpose chip for implementing logic circuits. It contains a collection of logic circuit elements that can be customized in different ways. A PLD can be viewed as a “black box” that contains logic gates and programmable switches, as illustrated in Figure B.24. The programmable switches allow the logic gates inside the PLD to be connected together to implement whatever logic circuit is needed.

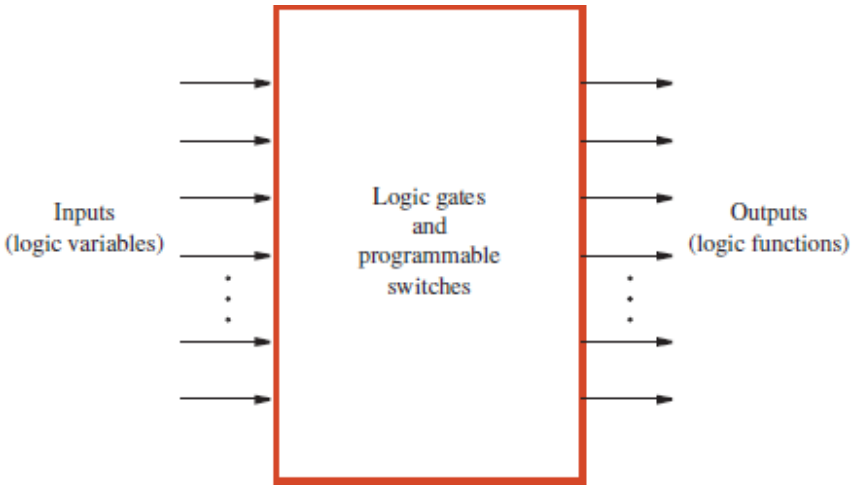


Figure B.24 Programmable logic device as a black box.

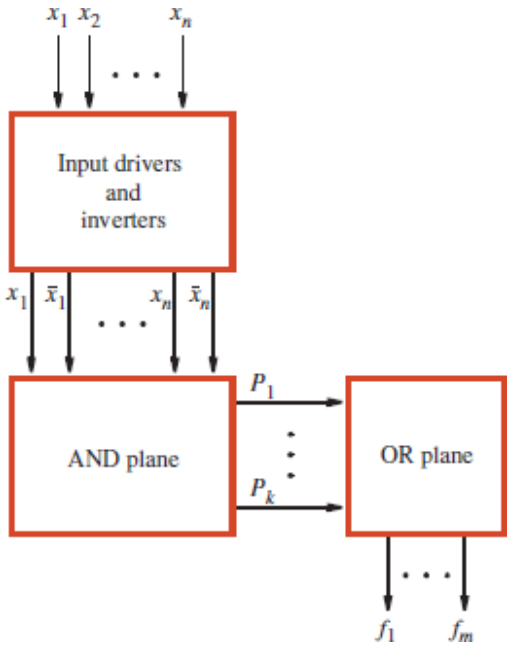


Figure B.25 General structure of a PLA.

B.6.1 PROGRAMMABLE LOGIC ARRAY (PLA)

Several types of PLDs are commercially available. The first developed was the *programmable logic array (PLA)*. The general structure of a PLA is depicted in Figure B.25. Based on the idea that logic functions can be realized in sum-of-products form, a PLA comprises a collection of AND gates that feeds a set of OR gates. As shown in the figure, the PLA's inputs x_1, …, x_n pass through a set of drivers (which provide both the true value and

complement of each input) into a circuit block called an *AND plane*, or *AND array*. The AND plane produces a set of product terms P_1, …, P_k. Each of these terms can be configured to implement any AND function of x_1, …, x_n. The product terms serve as the inputs to an *OR plane*, which produces the outputs f_1, …, f_m. Each output can be configured to realize any sum of P_1, …, P_k and hence any sum-of-products function of the PLA inputs.

A more detailed diagram of a small PLA is given in Figure B.26, which shows a PLA with three inputs, four product terms, and two outputs. Each AND gate in the AND plane has six inputs, corresponding to the true and complemented versions of the three input signals. Each connection to an AND gate is programmable; a signal that is connected to an AND gate is indicated with a wavy line, and a signal that is not connected to the gate is shown with a broken line. The circuitry is designed such that any unconnected AND-gate inputs do not affect the output of the AND gate. In commercially available PLAs, several methods of realizing the programmable connections exist. Detailed explanation of how a PLA can be built using transistors is given in Section B.10.

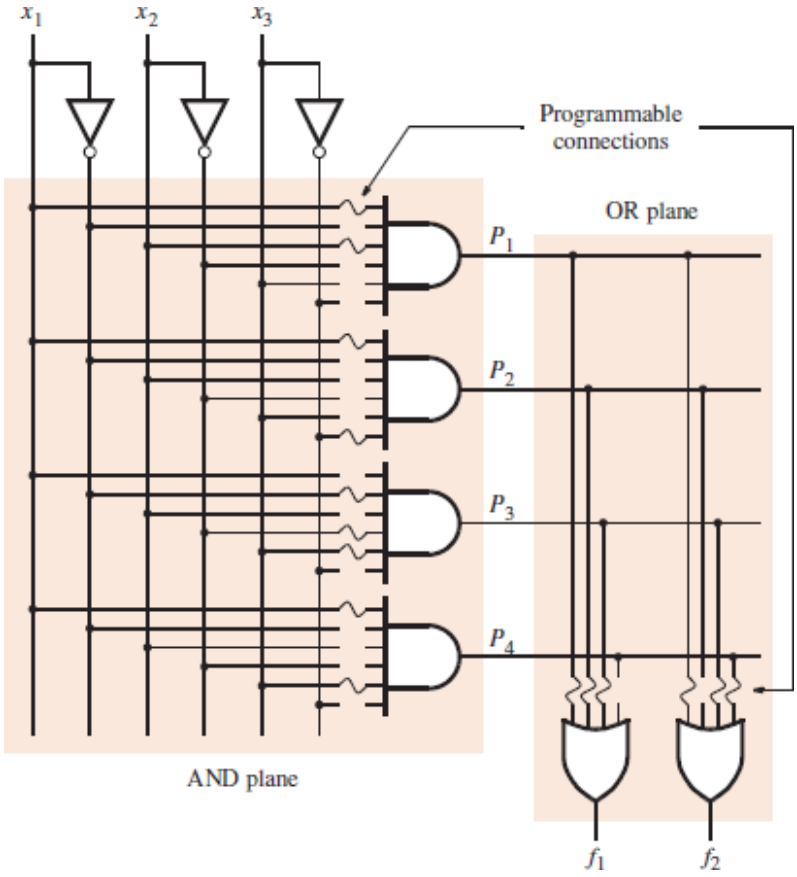


Figure B.26 Gate-level diagram of a PLA.

In Figure B.26 the AND gate that produces P_1 is shown connected to the inputs x_1 and x_2. Hence $P_1 = x_1\ x_2$. Similarly, $P_2 = x_1\bar{x}_3$, $P_3 = \bar{x}_1\bar{x}_2x_3$, and $P_4 = x_1x_3$. Programmable connections also exist for the OR plane. Output f_1 is connected to product terms P_1, P_2, and P_3. It therefore realizes the function $f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$. Similarly, output $f_2 = x_1x_2 + \bar{x}_1\bar{x}_2x_3 + x_1x_3$. Although Figure B.26 depicts the PLA programmed to implement the functions described above, by programming the AND and OR planes differently, each of the outputs f_1 and f_2 could implement various functions of x_1, x_2, and x_3. The only constraint on the functions that can be implemented is the size of the AND plane because it produces only four product terms. Commercially available PLAs come in larger sizes than we have shown here. Typical parameters are 16 inputs, 32 product terms, and eight outputs.

Although Figure B.26 illustrates clearly the functional structure of a PLA, this style of drawing is awkward for larger chips. Instead, it has become customary in technical literature to use the style shown in Figure B.27. Each AND gate is depicted as a single horizontal line attached to an AND-gate symbol. The possible inputs to the AND gate are drawn as vertical lines that cross the horizontal line. At any crossing of a vertical and horizontal line, a programmable connection, indicated by an X, can be made. Figure B.27 shows the programmable connections needed to implement the product terms in Figure B.26. Each OR gate is drawn in a similar manner, with a vertical line attached to an OR-gate symbol. The AND-gate outputs cross these lines, and corresponding programmable connections can be formed. The figure illustrates the programmable connections that produce the functions f_1 and f_2 from Figure B.26.

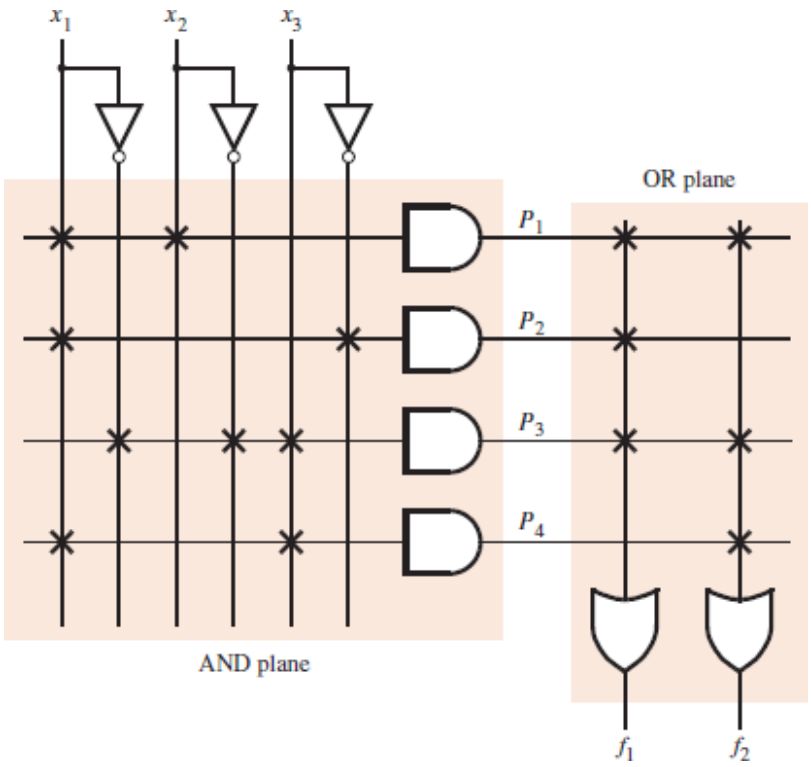


Figure B.27 Customary schematic for the PLA in Figure B.26.

The PLA is efficient in terms of the area needed for its implementation on an integrated circuit chip. For this reason, PLAs are often included as part of larger chips, such as microprocessors. In this case a PLA is created so that the connections to the AND and OR gates are fixed, rather than programmable. In Section B.10 we will show that both fixed and programmable PLAs can be created with similar structures.

B.6.2 PROGRAMMABLE ARRAY LOGIC (PAL)

In a PLA both the AND and OR planes are programmable. Historically, the programmable switches presented two difficulties for manufacturers of these devices: they were hard to fabricate correctly, and they reduced the speed-performance of circuits implemented in the PLAs. These drawbacks led to the development of a similar device in which the AND plane is programmable, but the OR plane is fixed. Such a chip is known as a *programmable array logic (PAL)* device.

An example of a PAL with three inputs, four product terms, and two outputs is given in Figure B.28. The product terms P_1 and P_2 are hardwired to one OR gate, and P_3 and P_4 are hardwired to the other OR gate. The PAL is shown

programmed to realize the two logic functions $f_1 = x_1x_2\bar{x}_3 + \bar{x}_1x_2x_3$ and $f_2 = \bar{x}_1\bar{x}_2 + x_1x_2x_3$. In comparison to the PLA in Figure B.27, the PAL offers less flexibility; the PLA allows up to four product terms per OR gate, whereas the OR gates in the PAL have only two inputs. To compensate for the reduced flexibility, PALs are manufactured in a range of sizes, with various numbers of inputs and outputs, and different numbers of inputs to the OR gates.

An example of a commonly used PAL is called the 22V10 [13], which is depicted in Figure B.29. There are 11 input pins that feed the AND plane, and an additional input that can also serve as a clock input. The OR gates are of variable size, ranging from 8 to 16 inputs.

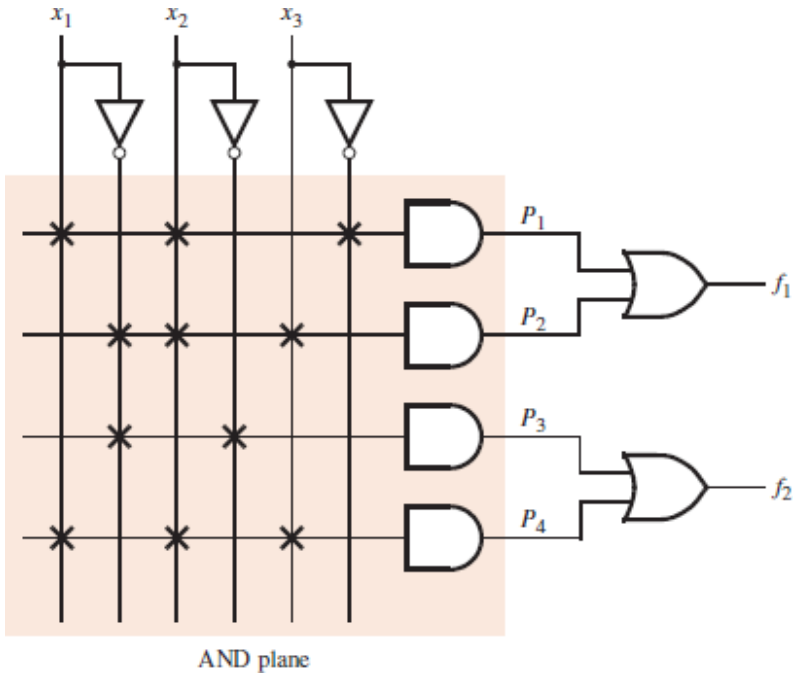


Figure B.28 An example of a PAL.

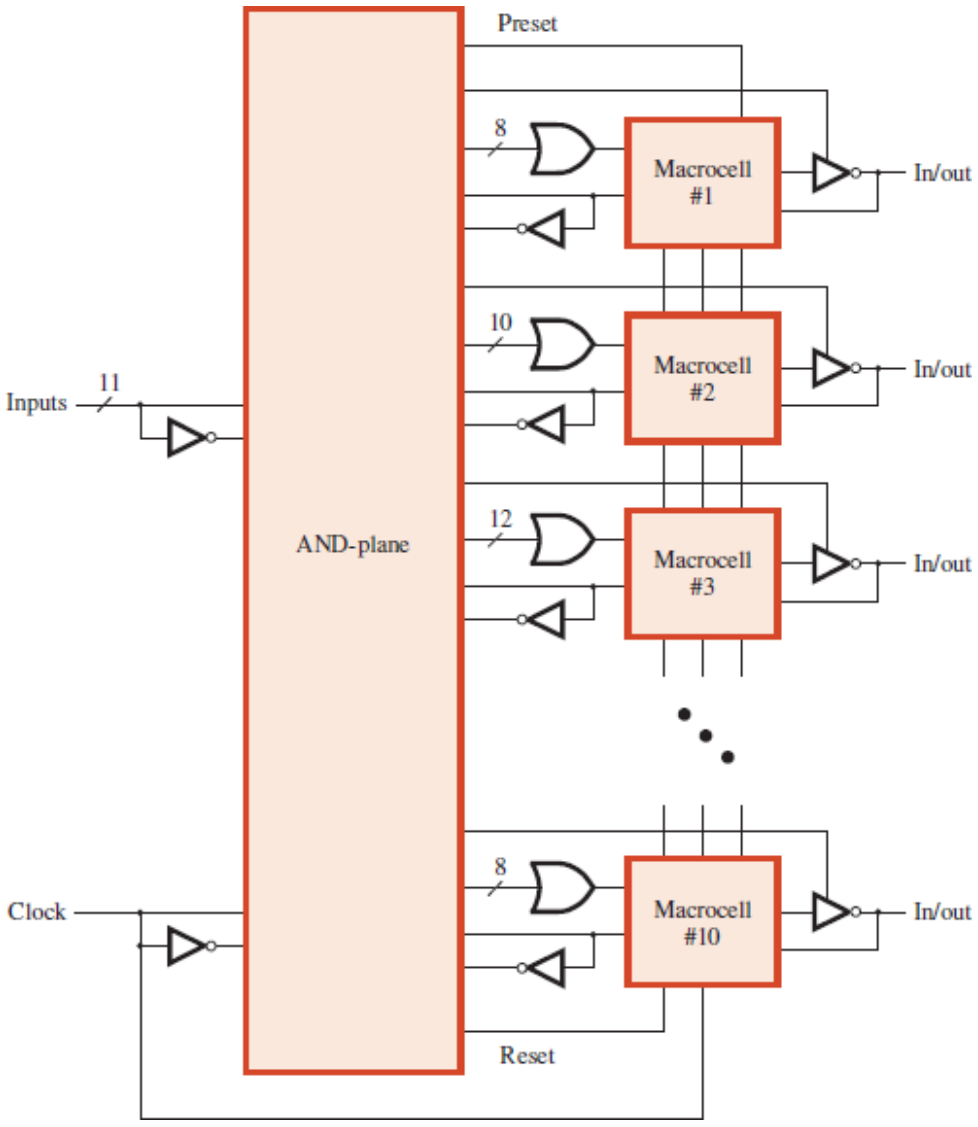


Figure B.29 The 22V10 PAL device.

Each output pin has a tri-state driver, which allows the pin to optionally be used as an input pin. Extra circuitry is connected to each OR gate and is referred to as a *macrocell*.

Figure B.30 shows one of the macrocells in the 22V10 PAL. It connects the OR gate to one input on an XOR gate, which feeds a D flip-flop. Since the other input to the XOR gate can be programmed to be 0 or 1, it can be used to complement the OR-gate output. A 2-to-1 multiplexer allows bypassing of the flip-flop, and the tri-state driver can be either permanently enabled or connected to a product term from the AND plane. Either the $\overline{Q}$ output from the flip-flop or the output of the tri-state driver can be connected to the AND

plane. If the tri-state driver is disabled, then the corresponding pin can be used as an input.

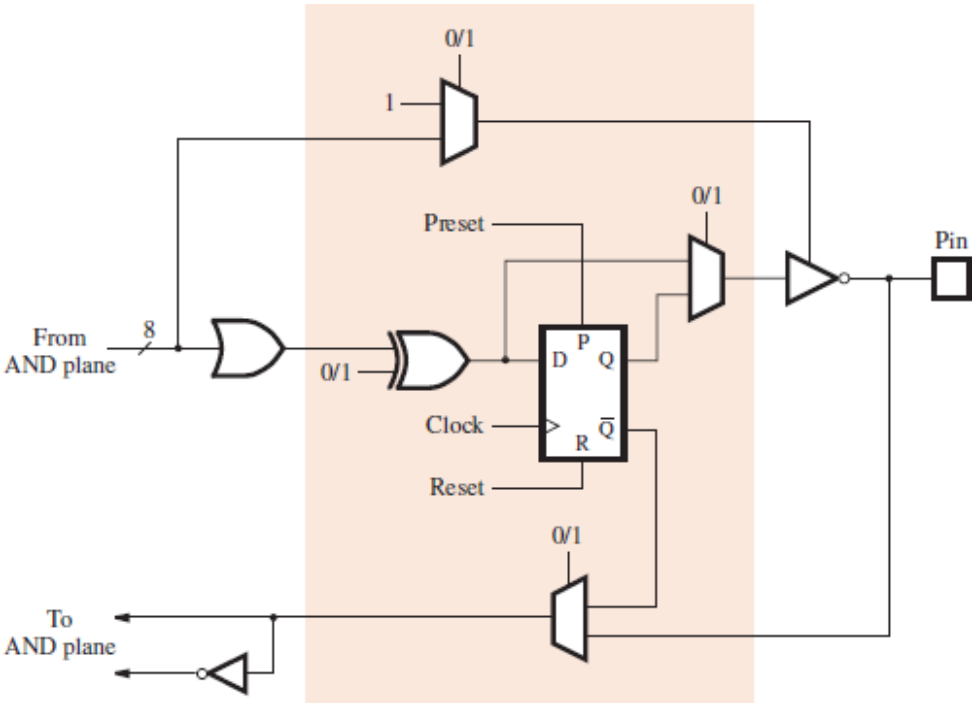


Figure B.30 The 22V10 macrocell.

PLAs, PALs, and other similar types of *simple PLDs (SPLDs)*, are manufactured by Microchip Technology. An interested reader can examine the information that this company provides on its web site.

B.6.3 PROGRAMMING OF PLAS AND PALS

In Figures B.27 and B.28, each connection between a logic signal in a PLA or PAL and the AND/OR gates is shown as an X. We describe how these switches are implemented using transistors in Section B.10. Users' circuits are implemented in the devices by *configuring*, or *programming*, these switches. Commercial chips contain a few thousand programmable switches, hence it is not feasible for a user of these chips to specify manually the desired programming state of each switch. Instead, CAD systems are employed for this purpose. CAD systems that support targeting of circuits to PLDs have the capability to automatically produce the necessary information for programming each of the switches in the device. A computer system that runs the CAD tools may be connected by a cable to a dedicated *programming unit*. Once the user has completed the design of a circuit, the CAD tools generate a

file, often called a *programming file* or *fuse map*, that specifies the state that each switch in the PLD should have, to realize correctly the designed circuit. The PLD is placed into the programming unit, and the programming file is transferred from the computer system. The programming unit places the chip into a special *programming mode* and configures each switch individually.

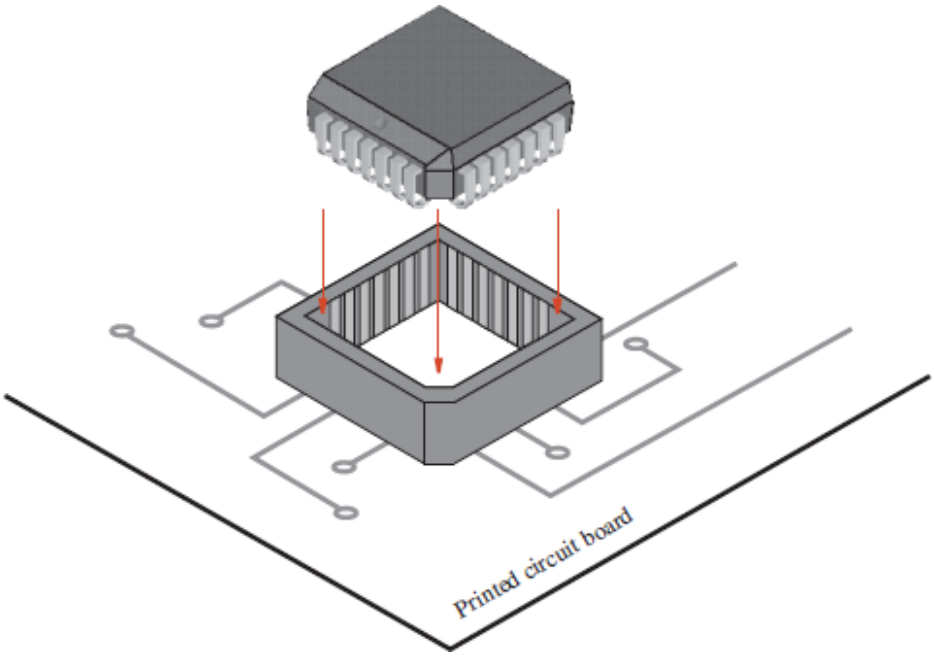


Figure B.31 A PLCC package with socket.

PLAs or PALs used as part of a logic circuit usually reside with other chips on a printed circuit board (PCB). The procedure described above assumes that the chip can be removed from the circuit board for programming in the programming unit. Removal is made possible by using a socket on the PCB, as illustrated in Figure B.31. Although PLAs and PALs are available in the DIP packages shown in Figure B.21*a*, they are also available in another popular type of package, called a *plastic-leaded chip carrier (PLCC)*, which is depicted in Figure B.31. On all four of its sides, the PLCC package has pins that "wrap around" the edges of the chip, rather than extending straight down as in the case of a DIP. The socket that houses the PLCC is attached by solder to the circuit board, and the PLCC is held in the socket by friction.

Instead of relying on a programming unit to configure a chip, it would be advantageous to be able to perform the programming while the chip is still attached to its circuit board. This method of programming is called *in-system*

programming (ISP). It is not usually provided for PLAs or PALs, but is available for the more sophisticated chips that are described below.

B.6.4 COMPLEX PROGRAMMABLE LOGIC DEVICES (CPLDS)

PLAs and PALs are useful for implementing a wide variety of small digital circuits. Each device can be used to implement circuits that do not require more than the number of inputs, product terms, and outputs that are provided in the particular chip. These chips are limited to fairly modest sizes, typically supporting a combined number of inputs plus outputs of not more than 32. For implementation of circuits that require more inputs and outputs, either multiple PLAs or PALs can be employed or else a more sophisticated type of chip, called a *complex programmable logic device (CPLD)*, can be used.

A CPLD comprises multiple circuit blocks on a single chip, with internal wiring resources to connect the circuit blocks. Each circuit block is similar to a PLA or a PAL; we will refer to the circuit blocks as *PAL-like blocks*. An example of a CPLD is given in Figure B.32. It includes four PAL-like blocks that are connected to a set of *interconnection wires*. Each PAL-like block is also connected to a subcircuit labeled *I/O block*, which is attached to a number of the chip's input and output pins.

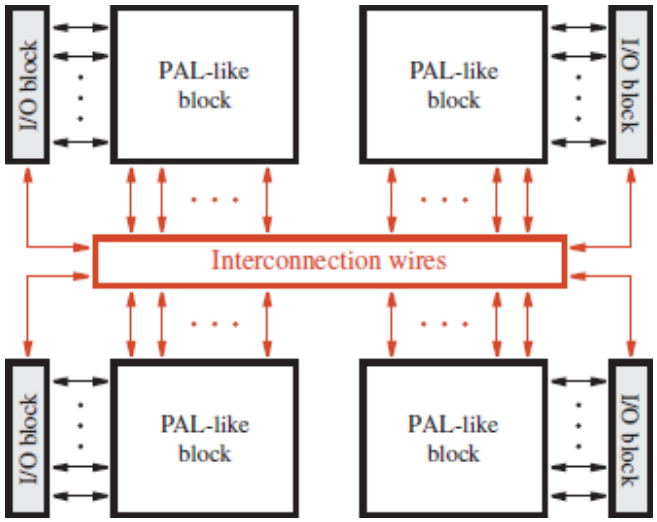


Figure B.32 Structure of a complex programmable logic device (CPLD).

Figure B.33 shows an example of the wiring structure and the connections to a PAL-like block in a CPLD. The PAL-like block includes 3 macrocells (real CPLDs typically have about 16 macrocells in a PAL-like block), each

consisting of a four-input OR gate (real CPLDs usually provide between 5 and 20 inputs to each OR gate). The OR-gate output is connected to an XOR gate; the other input to this gate can be programmably connected to 1 or 0; if 1, then the XOR gate complements the OR-gate output, and if 0, then the XOR gate has no effect. The macrocell also includes a flip-flop, a multiplexer, and a tri-state driver. Each tri-state driver is connected to a pin on the CPLD package. The tri-state driver allows each pin to be used either as an output from the CPLD or as an input.

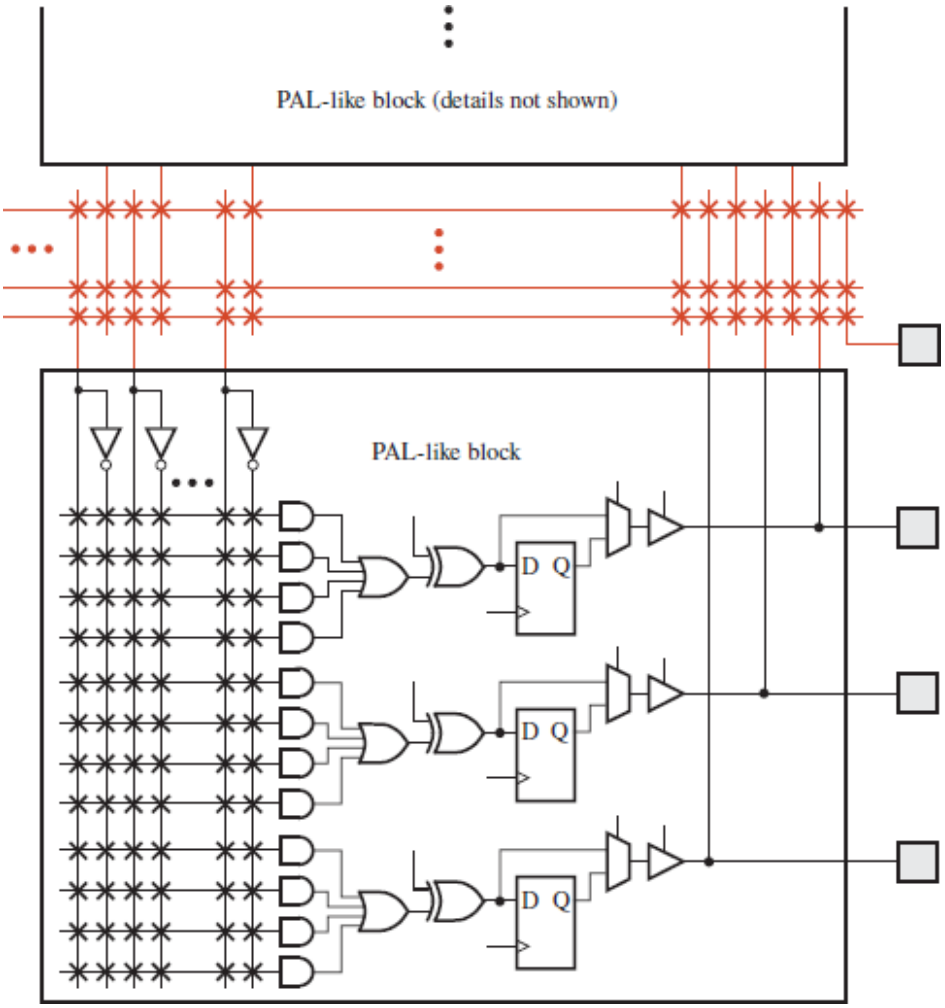
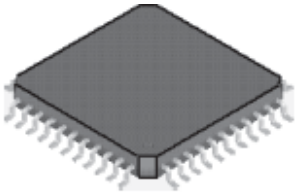


Figure B.33 A section of the CPLD in Figure B.32.

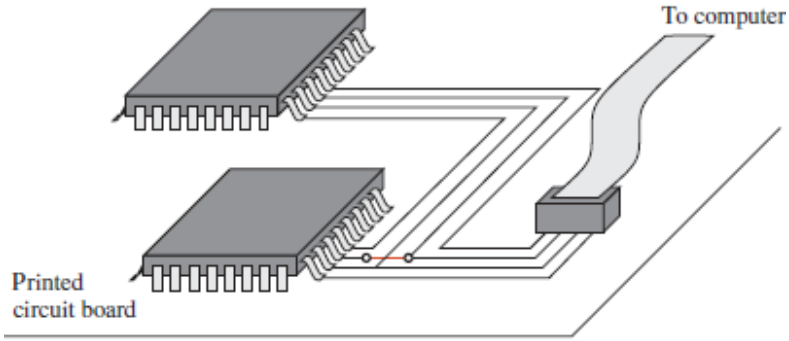
The interconnection wiring contains programmable switches that are used to connect the PAL-like blocks. Each of the horizontal wires can be connected to some of the vertical wires that it crosses. One detail to note is that when a pin is used as an input, the macrocell associated with that pin cannot be used and is therefore wasted. Some CPLDs include additional connections between

the macrocells and the interconnection wiring that avoids wasting macrocells in such situations.

Commercial CPLDs range in size from only 2 PAL-like blocks to more than 100 PAL-like blocks. They are available in a variety of packages, including the PLCC package that is shown in Figure B.31. Figure B.34*a* shows another type of package used to house CPLD chips, called a *quad flat pack* (QFP). Like a PLCC package, the QFP package has pins on all four sides, but whereas the PLCC's pins wrap around the edges of the package, the QFP's pins extend outward from the package, with a downward-curving shape. The QFP's pins are much thinner than those on a PLCC, which means that the package can support a larger number of pins; QFPs are available with more than 200 pins, whereas PLCCs are limited to fewer than 100 pins.



(a) CPLD in a Quad Flat Pack (QFP) package



(b) JTAG programming

Figure B.34 CPLD packaging and programming.

Most CPLDs contain the same type of programmable switches that are used in SPLDs, which are described in Section B.10. Programming of the switches may be accomplished using the same technique described in Section B.6.3, in which the chip is placed into a special-purpose programming unit. However, this programming method is rather inconvenient for large CPLDs for two reasons. First, large CPLDs may have more than 200 pins on the chip

package, and these pins are often fragile and easily bent. Second, to be programmed in a programming unit, a socket is required to hold the chip. Sockets for large QFP packages are very expensive; they may cost more than the CPLD device itself. For these reasons, CPLD devices usually support the ISP technique. A small connector is included on the PCB that houses the CPLD, and a cable is connected between that connector and a computer system. The CPLD is programmed by transferring the programming information generated by a CAD system through the cable, from the computer into the CPLD. The circuitry on the CPLD that allows this type of programming has been standardized by the IEEE and is usually called a *JTAG port*. It uses four wires to transfer information between the computer and the device being programmed. The term *JTAG* stands for Joint Test Action Group. Figure B.34*b* illustrates the use of a JTAG port for programming two CPLDs on a circuit board. The CPLDs are connected together so that both can be programmed using the same connection to the computer system. Once a CPLD is programmed, it retains the programmed state permanently, even when the power supply for the chip is turned off. This property is called *nonvolatile* programming.

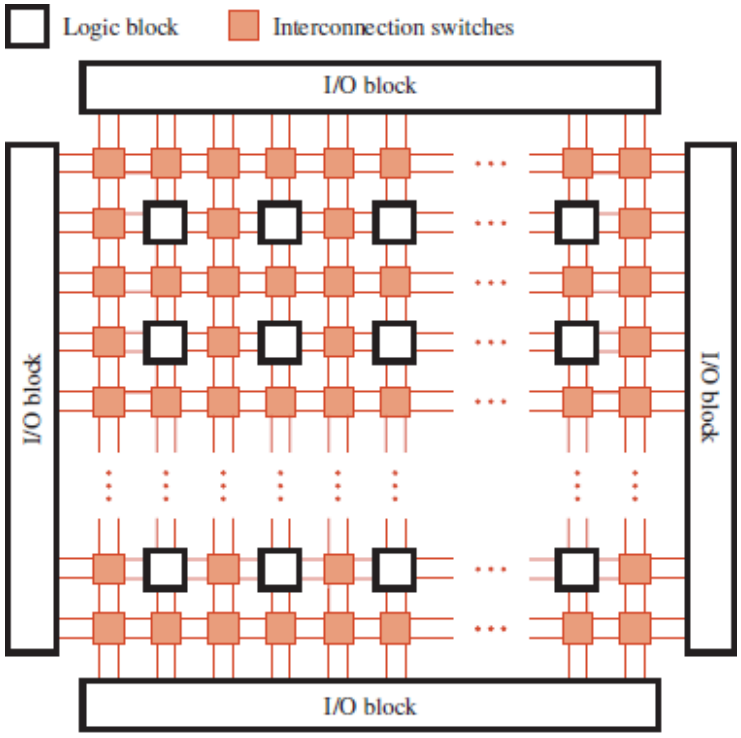
CPLDs are used for the implementation of many types of digital circuits. In industrial designs that employ some type of PLD device, CPLDs are used often, while SPLDs are becoming less common. A number of companies offer CPLDs, including Intel, Lattice, Microchip, and Xilinx. Information about these products is available on the internet.

B.6.5 FIELD-PROGRAMMABLE GATE ARRAYS

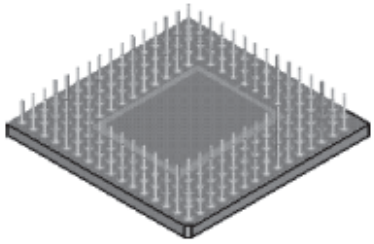
The types of chips described above, 7400 series, SPLDs, and CPLDs, are useful for implementation of small or moderately-sized logic circuits. One way to quantify a circuit's *size* is to assume that the circuit is to be built using only simple logic gates and then estimate how many of these gates are needed. A commonly used measure is the total number of two-input NAND gates that would be needed to build the circuit; this measure is often called the number of *equivalent gates*.

Using the equivalent-gates metric, the size of a 7400-series chip is simple to measure because each chip contains only simple gates. For SPLDs and CPLDs the typical measure used is that each macrocell represents about 20 equivalent gates. Thus a typical PAL that has eight macrocells can accommodate a circuit that needs up to about 160 gates, and a large CPLD

that has 500 macrocells can implement circuits of up to about 10,000 equivalent gates.



(a) General structure of an FPGA



(b) Pin grid array (PGA) package (bottom view)

Figure B.35 A field-programmable gate array (FPGA).

By modern standards, a logic circuit with 10,000 gates is not large. To implement larger circuits, it is convenient to use a different type of chip that has a larger logic capacity. A *field-programmable gate array (FPGA)* is a programmable logic device that supports implementation of relatively large logic circuits. FPGAs are quite different from SPLDs and CPLDs because FPGAs do not contain AND or OR planes. Instead, FPGAs provide *logic elements* for implementation of the required functions. The general structure

of an FPGA is illustrated in Figure B.35*a*. It contains three main types of resources: logic elements, I/O blocks for connecting to the pins of the package, and interconnection wires and switches. The logic elements are arranged in a two-dimensional array, and the interconnection wires are organized as horizontal and vertical *routing channels* between rows and columns of logic blocks. The routing channels contain wires and programmable switches that allow the logic elements to be interconnected in many ways. Figure B.35*a* shows two locations for programmable switches; the red boxes adjacent to logic elements hold switches that connect the logic element input and output terminals to the interconnection wires, and the red boxes that are diagonally between logic elements connect one interconnection wire to another (such as a vertical wire to a horizontal wire). Programmable connections also exist between the I/O blocks and the interconnection wires. The actual number of programmable switches and wires in an FPGA varies in commercially available chips.

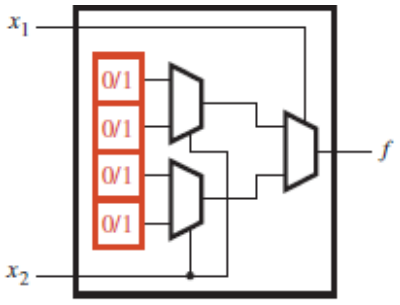
FPGAs can be used to implement logic circuits that comprise millions of equivalent logic gates. Examples of commercial FPGA products can be found on the websites provided by Intel, Microchip, Lattice, and Xilinx, which are the leading providers of these chips. FPGA chips are available in a variety of packages, including the PLCC and QFP packages described earlier. Figure B.35*b* depicts another type of package, called a *pin grid array (PGA).* A PGA package may have up to a few hundred pins in total, which extend straight outward from the bottom of the package, in a grid pattern. Another version of this packaging technology is known as the *ball grid array (BGA).* In a BGA the pins are small round balls, instead of posts. The advantage of BGA packages is that the pins are very small; hence more pins can be provided on a relatively small package.

Each logic element in an FPGA typically has a small number of inputs and outputs. A variety of FPGA products are on the market, featuring different types of logic elements. The most commonly used logic element is a *lookup table (LUT)*, which contains *storage cells* that are used to store the truth table of a logic function. Each cell is capable of holding a single logic value, either 0 or 1. LUTs of various *sizes* may be created, where the size is defined by the number of inputs. Figure B.36*a*s shows the structure of a small LUT. It has two inputs, x_1 and x_2, and one output, f. It is capable of implementing any logic function of two variables. Because a two-variable truth table has four rows, this LUT has four storage cells. One cell corresponds to the output value in each row of the truth table. The input variables x_1 and x_2 are used as the

select inputs of three multiplexers that provide the content of one of the four storage cells as the output of the LUT. We discuss storage cells in Section B.9.

To see how a logic function can be realized in the two-input LUT, consider the truth table in Figure B.36*b*. The function f_1 from this table can be stored in the LUT as illustrated in Figure B.36*c*. The arrangement of multiplexers in the LUT correctly realizes the function f_1. When $x_1 = x_2 = 0$, the output of the LUT is driven by the top storage cell, which represents the entry in the truth table for $x_1\ x_2 = 0\ 0$. Similarly, for all valuations of x_1 and x_2, the logic value stored in the storage cell corresponding to the entry in the truth table chosen by the particular valuation appears on the LUT output.

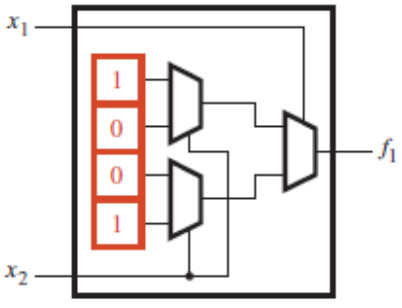
Figure B.37 shows a three-input LUT. It has eight storage cells because a three-variable truth table has eight rows. In commercial FPGA chips, LUTs usually have between four and six inputs, which require 16 and 64 storage cells, respectively. In addition to a LUT, FPGA logic elements usually include a flip-flop as indicated in Figure B.38.



(a) Circuit for a two-input LUT

x_1	x_2	f_1
0	0	1
0	1	0
1	0	0
1	1	1

(b) $f_1 = \bar{x}_1\bar{x}_2 + x_1x_2$



(c) Storage cell contents in the LUT

Figure B.36 A two-input lookup table (LUT).

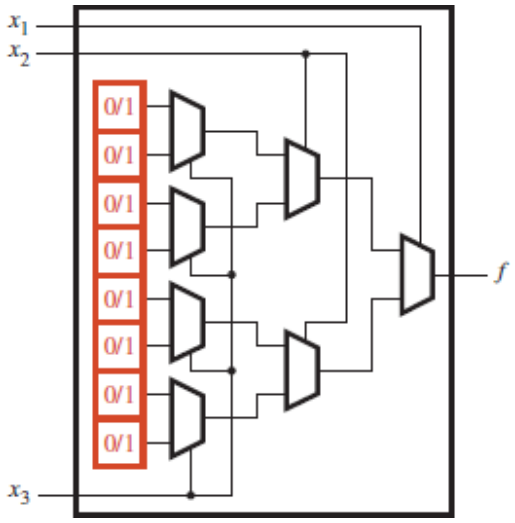


Figure B.37 A three-input LUT.

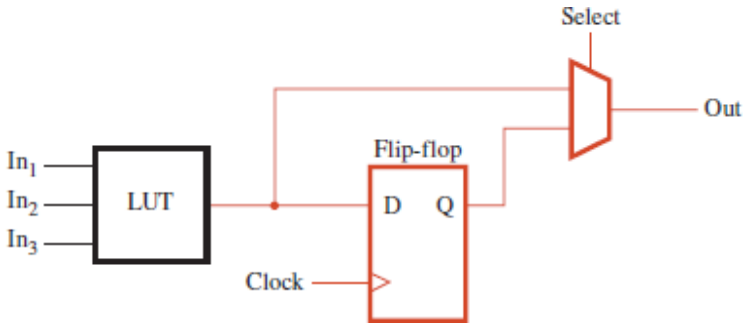


Figure B.38 Inclusion of a flip-flop in an FPGA logic element.

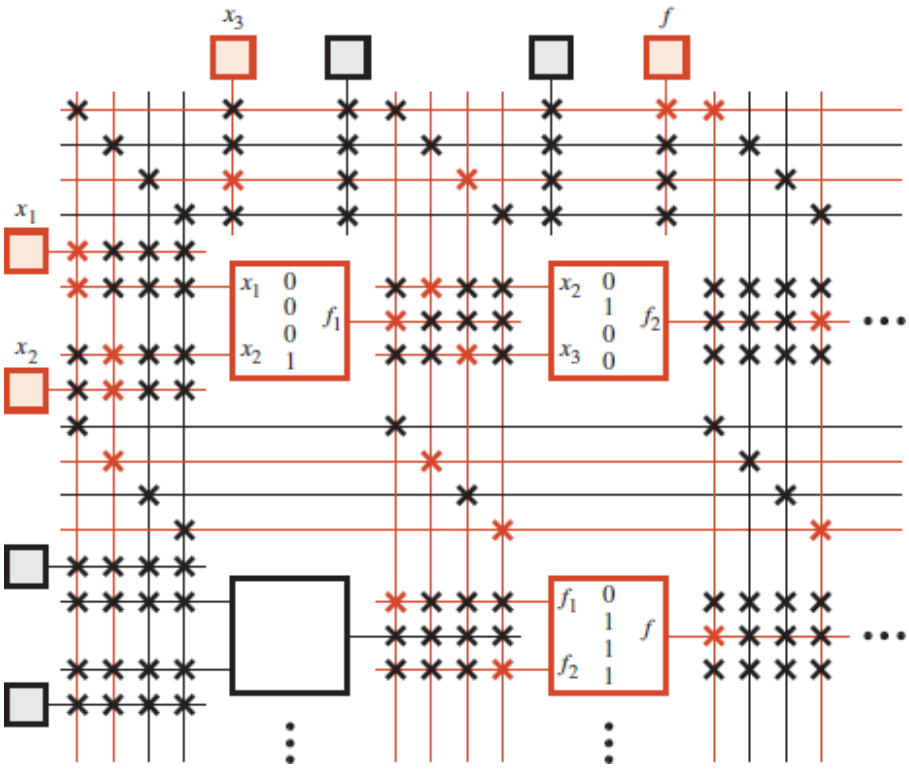


Figure B.39 A section of a programmed FPGA.

For a logic circuit to be realized in an FPGA, each logic function in the circuit must be small enough to fit within a single logic element. In practice, a user's circuit is automatically translated into the required form by using CAD tools (see Chapter 10). When a circuit is implemented in an FPGA, the logic elements are programmed to realize the necessary functions and the routing channels are programmed to make the required interconnections between logic elements. FPGAs are configured by using the ISP method, which we explained in Section B.6.4. The storage cells in the LUTs in an FPGA are *volatile*, which means that they lose their stored contents whenever the power supply for the chip is turned off. Hence the FPGA has to be programmed every time power is applied. Often a small memory chip that holds its data permanently, called a *programmable read-only memory (PROM)*, is included on the circuit board that houses the FPGA. The storage cells in the FPGA are loaded automatically from the PROM when power is applied to the chips.

A small FPGA that has been programmed to implement a circuit is depicted in Figure B.39. The FPGA has two-input LUTs, and there are four wires in each routing channel. The figure shows the programmed states of both the logic elements and wiring switches in a section of the FPGA. Programmable wiring switches are indicated by an X. Each switch shown in red is turned on

and makes a connection between a horizontal and vertical wire. The switches shown in black are turned off. We describe how the switches are implemented by using transistors in Section B.10.1. The truth tables programmed into the logic elements in the top row of the FPGA correspond to the functions $f_1 = x_1 x_2$ and $f_2 = \bar{x}_2 x_3$. The logic element in the bottom right of the figure is programmed to produce $f = f_1 + f_2 = x_1 x_2 + \bar{x}_2 x_3$.

B.7 Custom Chips, Standard Cells, and Gate Arrays

The programmable switches in a PLD provide the important benefit of user programmability, but they consume a significant amount of space on the chip and lead to increased cost. They also result in a reduction in the speed of operation of circuits, and an increase in power consumption. In this section we will introduce some high-capacity integrated circuit technologies that do not contain programmable switches.

To provide the largest number of logic gates, highest circuit speed, or lowest power, a so-called *custom chip* can be manufactured. Whereas a PLD is prefabricated, containing logic gates and programmable switches that are programmed to realize a user's circuit, a custom chip is created from scratch. The designer of a custom chip has complete flexibility to decide the size of the chip, the number of transistors the chip contains, the placement of each transistor on the chip, and the way the transistors are connected together. The process of defining exactly where on the chip each transistor and wire is situated is called *chip layout*. For a custom chip the designer may create any layout that is desired. A custom chip requires a large amount of design effort and is therefore expensive. Consequently, such chips are produced only when standard parts like FPGAs do not meet the requirements. To justify the expense of a custom chip, the product being designed must be expected to sell in sufficient quantities to recoup the cost. Two examples of products that are usually realized with custom chips are microprocessors and memory chips.

In situations where the chip designer does not need complete flexibility for the layout of each individual transistor in a custom chip, some of the design effort can be avoided by using a technology known as *standard cells*. Chips made using this technology are often called *application-specific integrated circuits (ASICs)*. An example of a small section of an ASIC is depicted in

Figure B.40. CAD tools are used to automatically place the standard cells into rows and to create interconnecting wires in the *routing channels* between the rows. Many variants of standard cells are usually provided, including basic logic gates, latches, flip-flops, and so on. The available standard cells are prebuilt and are stored in a library that can be accessed by the designer. In Figure B.40 the interconnection wires are drawn in two colors. This scheme is used because metal wires can be created on integrated circuits in multiple *layers*, which makes it possible for two wires to cross one another without creating a short circuit. The red wires represent one layer of metal wires, and the black wires are a different layer. Each red square represents a hard-wired connection (called a *via*) between a wire on one layer and a wire on the other layer. In current technology it is possible to have ten or more layers of metal wiring. Some of the metal layers can be placed on top of the transistors in the logic gates, resulting in a more efficient chip layout.

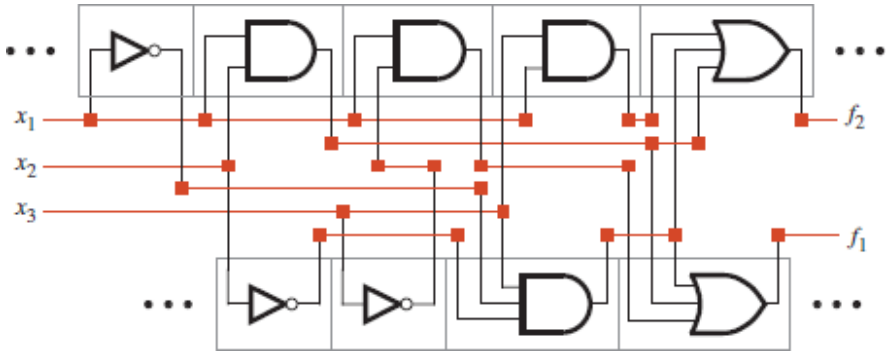


Figure B.40 A section of two rows in a standard-cell chip.

Like a custom chip, a standard-cell chip is created from scratch according to a user's specifications. The circuitry shown in Figure B.40 implements the two logic functions that we realized in a PLA in Figure B.26, namely, $f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$ and $f_2 = x_1x_2 + \bar{x}_1\bar{x}_2x_3 + x_1x_3$. Because of the expense involved, a standard-cell chip would never be created for a small circuit such as this one, and thus the figure shows only a portion of a much larger chip. The layout of individual gates (standard cells) is predesigned and fixed. The chip layout can be created automatically by CAD tools because of the regular arrangement of the logic gates (cells) in rows. A typical chip has many long rows of logic gates with a large number of wires between each pair of rows. The I/O blocks around the periphery connect to the pins of the chip package, such as a QFP, PGA, or BGA package.

Another technology, similar to standard cells, is the *gate-array* technology. In a gate array parts of the chip are prefabricated, and other parts are custom fabricated for a particular user's circuit. This concept exploits the fact that integrated circuits are fabricated in a sequence of steps, some steps to create transistors and other steps to create wires to connect the transistors together. In gate-array technology, the manufacturer performs most of the fabrication steps, typically those involved in the creation of the transistors, without considering the requirements of a user's circuit. This process results in a silicon wafer of partially finished chips, called the gate-array *template*. Later the template is modified, usually by fabricating wires that connect the transistors together, to create a user's circuit in each finished chip. The gate-array approach provides cost savings in comparison to the custom-chip approach because the gate-array manufacturer can amortize the cost of chip fabrication over a large number of template wafers, all of which are identical. Many variants of gate-array technology exist. Some have relatively large logic cells, while others are configurable at the level of a single transistor.

An example of a gate-array template is given in Figure B.41. The gate array contains a two-dimensional array of logic cells. The chip's general structure is similar to a standard-cell chip except that in the gate array all logic cells are identical. Although the types of logic cells used in gate arrays vary, one common example is a two- or three-input NAND gate. In some gate arrays empty spaces exist between the rows of logic cells to accommodate the wires that will be added later to connect the logic cells together. However, most gate arrays do not have spaces between rows of logic cells, and the interconnection wires are fabricated on top of the logic cells. This design is possible because, as discussed for Figure B.40, metal wires can be created on a chip in multiple layers. This approach is known as the *sea-of-gates* technology. Figure B.42 depicts a small section of a gate array that has been customized to implement the logic function $f = x_2\bar{x}_3 + x_1x_3$. As we showed in Section 2.7, it is easy to verify that this circuit with only NAND gates is equivalent to the AND-OR form of the circuit.

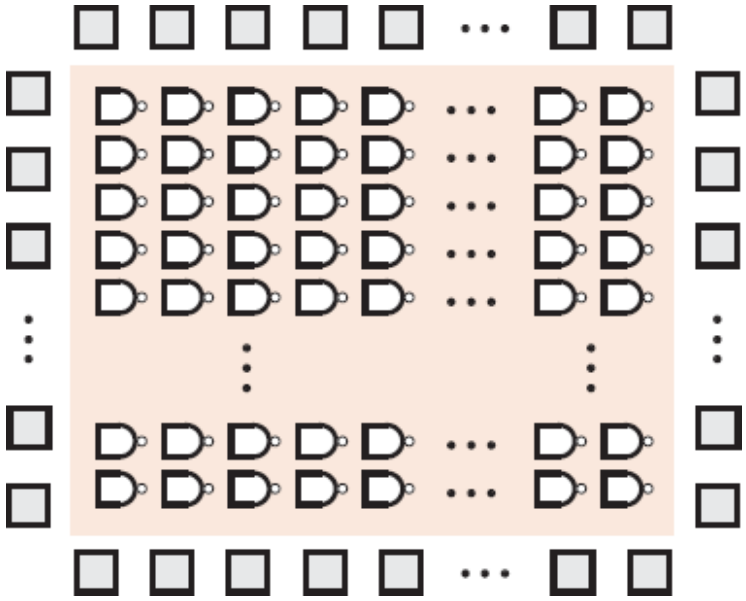


Figure B.41 A sea-of-gates gate array.

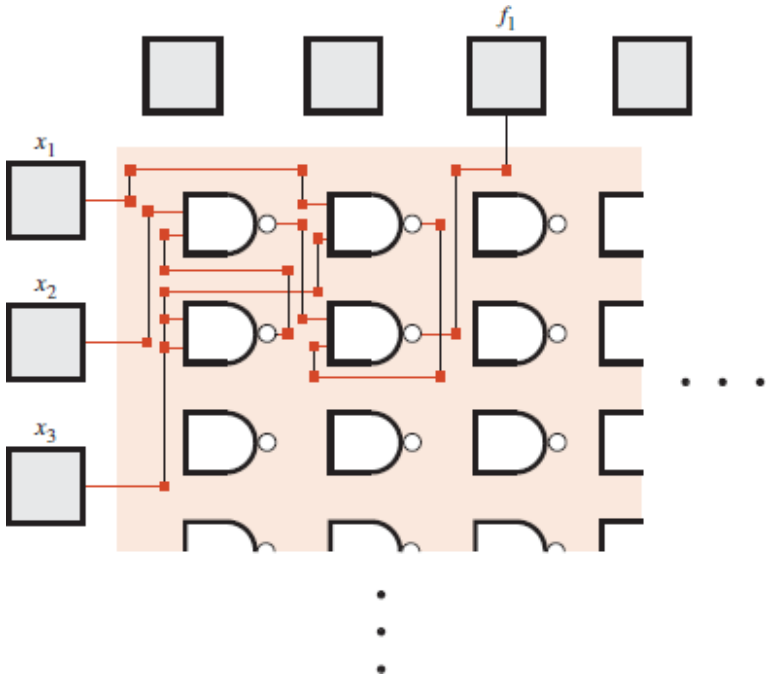


Figure B.42 The logic function $f_1 = x_2\bar{x}_3 + x_1x_3$ in the gate array of Figure B.41.

B.8 Practical Aspects

In this section we provide more detailed information on several aspects of digital circuits. We describe how transistors are fabricated in silicon and give a detailed explanation of how transistors operate. We discuss the robustness of logic circuits and discuss the important issues of signal propagation delays and power dissipation in logic gates.

B.8.1 MOSFET FABRICATION AND BEHAVIOR

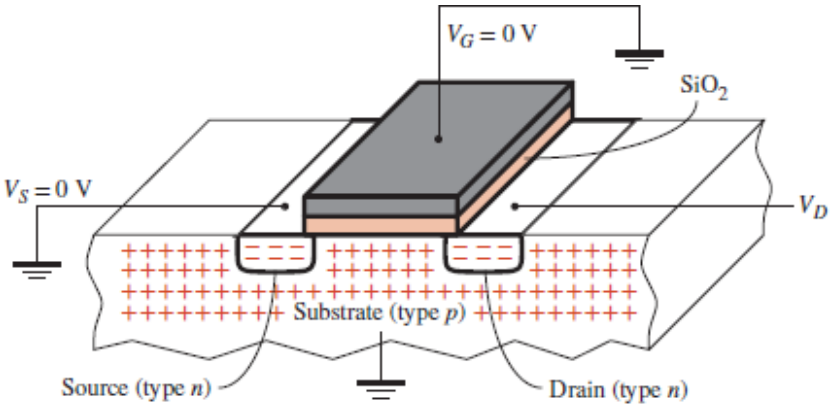
To understand the operation of NMOS and PMOS transistors, we need to consider how they are built in an integrated circuit. Integrated circuits are fabricated on silicon wafers. A silicon wafer is usually 6, 8, or 12 inches in diameter and is somewhat similar in appearance to an audio compact disc (CD). Many integrated circuit chips are fabricated on one wafer, and the wafer is then cut to provide the individual chips.

Silicon is an *electrical semiconductor*, which means that it can be manipulated such that it sometimes conducts electrical current and at other times does not. A transistor is fabricated by creating areas in the silicon substrate that have an excess of either positive or negative electrical charge. Negatively charged areas are called *type n*, and positively charged areas are *type p*. Figure B.43 illustrates the structure of an NMOS transistor. It has type *n* silicon for both the source and drain terminals, and type *p* for the substrate terminal. Metal wiring is used to make electrical connections to the source and drain terminals.

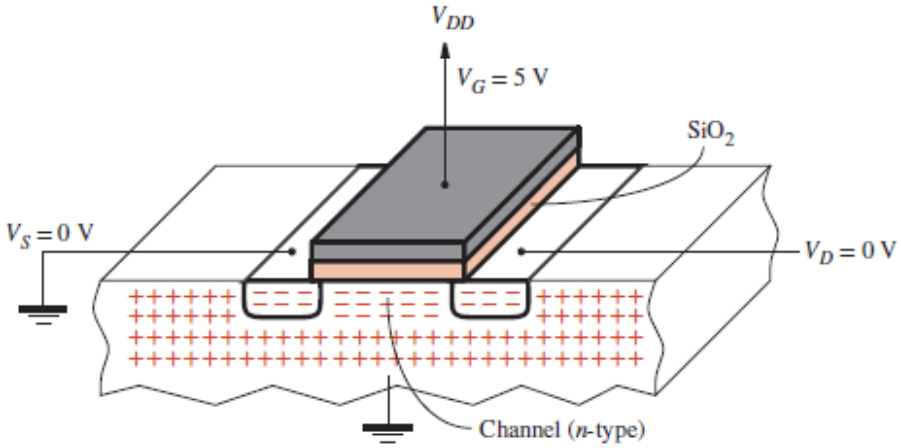
When MOSFETs were invented, the gate terminal was made of metal. Now a material known as *polysilicon* is used. Like metal, polysilicon is a conductor. The gate is electrically isolated from the rest of the transistor by a layer of silicon dioxide (SiO_2), which is a type of glass that acts as an electrical insulator between the gate terminal and the substrate of the transistor. The transistor's operation is governed by electrical fields caused by voltages applied to its terminals, as discussed below.

In Figure B.43 the voltage levels applied at the source, gate, and drain terminals are labeled V_S, V_G, and V_D, respectively. Consider first the situation depicted in Figure B.43*a* in which both the source and gate are connected to *Gnd* ($V_S = V_G = 0$ V). The type *n* source and type *n* drain are isolated from one another by the type *p* substrate. In electrical terms two diodes exist between the source and drain. One diode is formed by the *p*–*n* junction between the substrate and source, and the other diode is formed by the *p*–*n* junction between the substrate and drain. These back-to-back diodes represent a very

high resistance (about 10^{12} Ω [1]) between the drain and source that prevents current flow. We say that the transistor is *turned off*, or *cut off*, in this state.



(a) When $V_{GS} = 0$ V, the transistor is off



(b) When $V_{GS} = 5$ V, the transistor is on

Figure B.43 Physical structure of an NMOS transistor.

Next consider the effect of increasing the voltage at the gate terminal with respect to the voltage at the source. Let V_{GS} represent the gate-to-source voltage. If V_{GS} is greater than a certain minimum positive voltage, called the *threshold voltage* V_T, then the transistor changes from an open switch to a closed switch, as explained below. The exact level of V_T depends on many factors, but it is typically about 0.2 V_{DD}.

The transistor's state when $V_{GS} > V_T$ is illustrated in Figure B.43*b*. The gate terminal is connected to V_{DD}, resulting in V_{GS} = 5 V. The positive voltage on

the gate attracts free electrons that exist in the type *n* source terminal, as well as in other areas of the transistor, toward the gate. Because the electrons cannot pass through the layer of glass under the gate, they gather in the region of the substrate between the source and drain, which is called the *channel*. This concentration of electrons *inverts* the silicon in the area of the channel from type *p* to type *n*, which effectively connects the source and the drain. The size of the channel is determined by the length and width of the gate. The channel length L is the dimension of the gate between the source and drain, and the channel width W is the other dimension. The channel can also be thought of as having a depth, which is dependent on the applied voltages at the source, gate, and drain.

No current can flow through the gate node of the transistor, due to the layer of glass that insulates the gate from the substrate. A current I_D may flow from the drain node to the source. For a fixed value of $V_{GS} > V_T$, the value of I_D depends on the voltage applied across the channel V_{DS}. If $V_{DS} = 0\,\text{V}$, then no current flows. As V_{DS} is increased, I_D increases approximately linearly with the applied V_{DS}, as long as V_D is sufficiently small to provide at least V_T volts across the drain end of the channel, that is $V_{GD} > V_T$. In this range of voltages, namely, $0 < V_{DS} < (V_{GS} - V_T)$, the transistor is said to operate in the *triode region*, also called the *linear region*. The relationship between voltage and current is approximated by the equation

$$I_D = k'_n \frac{W}{L}\left[(V_{GS} - V_T)V_{DS} - \frac{1}{2}V_{DS}^2\right] \qquad \textbf{[B.1]}$$

The symbol k'_n is called the *process transconductance parameter*. It is a constant that depends on the technology being used and has the units A/V^2.

As V_D is increased, the current flow through the transistor increases, as given by Equation B.1, but only to a certain point. When $V_{DS} = V_{GS} - V_T$, the current reaches its maximum value. For larger values of V_{DS}, the transistor is no longer operating in the triode region. Since the current is at its saturated (maximum) value, we say that the transistor is in the *saturation region*. The current is now independent of V_{DS} and is given by the expression

$$I_D = \frac{1}{2}k'_n \frac{W}{L}(V_{GS} - V_T)^2 \qquad \textbf{[B.2]}$$

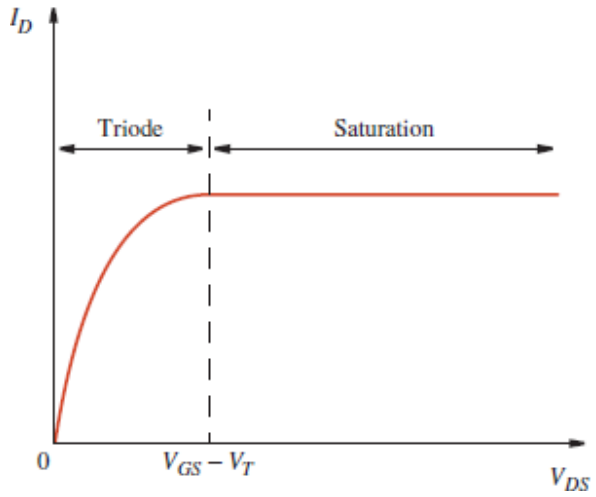


Figure B.44 The current-voltage relationship in the NMOS transistor.

Figure B.44 shows the shape of the current-voltage relationship in the NMOS transistor for a fixed value of $V_{GS} > V_T$. The figure indicates the point at which the transistor leaves the triode region and enters the saturation region, which occurs at $V_{DS} = V_{GS} - V_T$.

Example B.3

Assume the values $k_n' = 60\ \mu A/V^2$, $W/L = 2.0\ \mu m/0.5\ \mu m$, $V_S = 0$ V, $V_G = 5$ V, and $V_T = 1$ V. If V_D=2.5 V, the current in the transistor is given by Equation B.1 as $I_D \approx$ 1.7 mA. If V_D=5 V, the saturation current is calculated using Equation B.2 as $I_D \approx 2$ mA.

The PMOS Transistor

The behavior of PMOS transistors is the same as for NMOS except that all voltages and currents are reversed. The source terminal of the PMOS transistor is the terminal with the higher voltage level (recall that for an NMOS transistor the source terminal is the one with the lower voltage level), and the threshold voltage required to turn the transistor on has a negative value. PMOS transistors have the same physical construction as NMOS transistors except that wherever the NMOS transistor has type *n* silicon, the PMOS transistor has type *p*, and vice versa. For a PMOS transistor the equivalent of Figure B.43*a* is to connect both the source and gate nodes to

V_{DD}, in which case the transistor is turned off. To turn the PMOS transistor on, equivalent to Figure B.43*b*, we would set the gate node to *Gnd*, resulting in V_{GS} = -5 V.

Because the channel is type *p* silicon, instead of type *n*, the physical mechanism for current conduction in PMOS transistors is different from that in NMOS transistors. A detailed discussion of this concept is beyond the scope of this book, but one implication has to be mentioned. Equations B.1 and B.2 use the parameter k'_n. The corresponding parameter for a PMOS transistor is k'_p, but current flows more readily in type *n* silicon than in type *p*, with the result that in a typical technology $k'_p \approx 0.4 \times k'_n$. For a PMOS transistor to have current capacity equal to that of an NMOS transistor, we must use *W*/*L* of about two to three times larger in the PMOS transistor. In logic gates the sizes of NMOS and PMOS transistors are usually chosen to account for this factor.

B.8.2 MOSFET ON-RESISTANCE

In Section B.1 we considered MOSFETs as ideal switches that have infinite resistance when turned off and zero resistance when on. The actual resistance in the channel when the transistor is turned on, referred to as the *on-resistance*, is given by V_{DS}/I_D. Using Equation B.1 we can calculate the on-resistance in the triode region, as shown below.

Example B.4

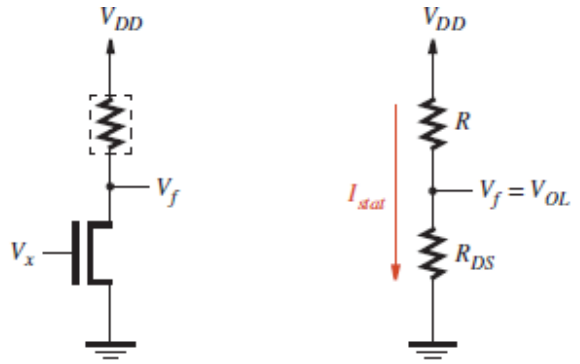
Consider a CMOS inverter in which the input voltage V_x is equal to 5 V. The NMOS transistor is turned on, and the output voltage V_f is close to 0 V. Hence V_{DS} for the NMOS transistor is close to zero and the transistor is operating in the triode region. In the curve in Figure B.44, the transistor is operating at a point very close to the origin. Although the value of V_{DS} is small, it is not exactly zero. In the next section we explain that V_{DS} would typically be about 0.1 mV. Hence the current I_D is not exactly zero; it is defined by Equation B.1. In this equation we can ignore the term involving V^2_{DS} because V_{DS} is small. In this case the on-resistance is approximated by

$$R_{DS} = V_{DS}/I_D = 1/\left[k'_n \frac{W}{L}(V_{GS} - V_T)\right] \qquad \text{[B.3]}$$

Assuming the values $k'_n = 60\ \mu A/V^2$, $W/L = 2.0\ \mu m/0.5\ \mu m$, $V_{GS} = 5$ V, and $V_T = 1$ V, we get $R_{DS} \approx 1$ kΩ.

B.8.3 VOLTAGE LEVELS IN LOGIC GATES

In Figure B.1 we showed that the logic values are represented by a range of voltage levels. We now consider the issue of voltage levels more carefully.



(a) NMOS NOT gate (b) $V_x = 5$ V

Figure B.45 Voltage levels in the NMOS inverter.

The high and low voltage levels in a logic family can be characterized by the operation of its basic inverter. Figure B.45*a* reproduces the circuit in Figure B.5 for an inverter built with NMOS technology. When V_x=0 V, the NMOS transistor is turned off. No current flows; hence V_f= 5 V. When V_x = V_{DD}, the NMOS transistor is turned on. To calculate the value of V_f, we can represent the NMOS transistor by a resistor with the value R_{DS}, as illustrated in Figure B.45*b*. Then V_f is given by the voltage divider

$$V_f = V_{DD}\frac{R_{DS}}{R_{DS} + R}$$

Example B.5

Assume that R=25 $k\Omega$. Using the result from Example B.4, R_{DS}=1 $k\Omega$, which gives $V_f \approx 0.2$ V.

As indicated in Figure B.45*b*, a current I_{stat} flows through the NMOS inverter under the static condition $V_x = V_{DD}$. This current is given by

$$I_{stat} = V_f / R_{DS} = 0.2\ \text{V} / 1\ \text{k}\Omega = 0.2\ \text{mA}$$

This static current has important implications, which we discuss in Section B.8.6.

In practice, the pull-up device R is usually implemented using a PMOS transistor. Such circuits are referred to as *pseudo-NMOS circuits*. They are fully compatible with CMOScircuits; hence a single chip may contain both CMOS and pseudo-NMOS gates. Example B.17 discusses a circuit for a pseudo-NMOS inverter and shows how to calculate its output voltage levels.

The CMOS Inverter

It is customary to use the symbols V_{OH} and V_{OL} to characterize the voltage levels in a logic circuit. The meaning of V_{OH} is the voltage produced when the output is high. Similarly, V_{OL} refers to the voltage produced when the output is low. As discussed above, in the NMOS inverter $V_{OH} = V_{DD}$ and V_{OL} is about 0.2 V.

Consider again the CMOS inverter in Figure B.12*a*. Its output-input voltage relationship is summarized by the *voltage transfer characteristic* shown in Figure B.46. The curve gives the steady-state value of V_f for each value of V_x. When $V_x = 0$ V, the NMOS transistor is off. No current flows; hence $V_f = V_{OH} = V_{DD}$. When $V_x = V_{DD}$, the PMOS transistor is off, no current flows, and $V_f = V_{OL} = 0$ V. For completeness we should mention that even when a transistor is turned off, a small current, called the *leakage* current, may flow through it. This current has a slight effect on V_{OH} and V_{OL}. For example, a typical value of V_{OL} is 0.1 mV, rather than 0 V [1].

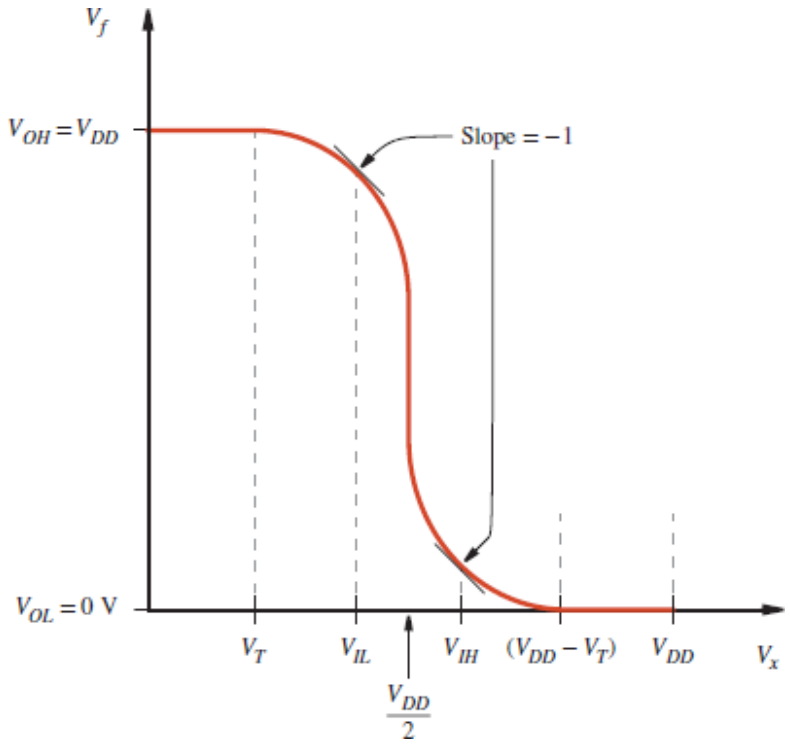


Figure B.46 The voltage transfer characteristic for the CMOS inverter.

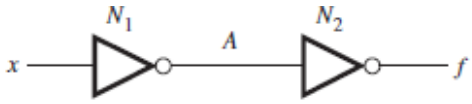
Figure B.46 includes labels at the points where the output voltage begins to change from high to low, and vice versa. The voltage V_{IL} represents the point where the output voltage is high and the slope of the curve equals -1. This voltage level is defined as the maximum input voltage level that the inverter will interpret as low, hence producing a high output. Similarly, the voltage V_{IH}, which is the other point on the curve where the slope equals -1, is the minimum input voltage level that the inverter will interpret as high, hence producing a low output. The parameters V_{OH}, V_{OL}, V_{IL}, and V_{IH} are important for quantifying the robustness of a logic family, as discussed below.

B.8.4 NOISE MARGIN

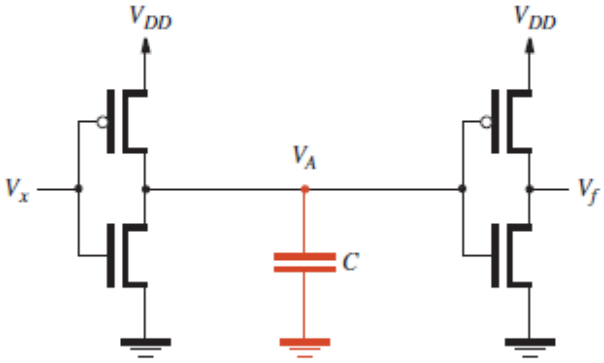
Consider the two NOT gates shown in Figure B.47*a*. Let us refer to the gates on the left and right as N_1 and N_2, respectively. Electronic circuits are constantly subjected to random perturbations, called *noise*, which can alter the output voltage levels produced by the gate N_1. It is essential that this noise not cause the gate N_2 to misinterpret a low logic value as a high one, or vice versa. Consider the case where N_1 produces its low voltage level V_{OL}. The presence

of noise may alter the voltage level, but as long as it remains less than V_{IL}, it will be interpreted correctly by N_2. The ability to tolerate noise without affecting the correct operation of the circuit is known as *noise margin*. For the low output voltage, we define the low *noise margin* as

$$NM_L = V_{IL} - V_{OL}$$



(a) A NOT gate driving another NOT gate



(b) The capacitive load at node A

Figure B.47 Parasitic capacitance in integrated circuits.

A similar situation exists when N_1 produces its high output voltage V_{OH}. Any existing noise in the circuit may alter the voltage level, but it will be interpreted correctly by N_2 as long as the voltage is greater than V_{IH}. The high noise margin is defined as

$$NM_H = V_{OH} - V_{IH}$$

Example B.6

For a given technology the voltage transfer characteristic of the basic inverter determines the levels V_{OH}, V_{OL}, V_{IL}, and V_{IH}. For CMOS we showed in Figure B.46 that $V_{OH} = V_{DD}$ and $V_{OL} = 0$ V. By finding the two points where the slope of the voltage transfer characteristic is equal to -1, it can be shown [1] that $V_{IL} \cong \frac{1}{8}(3V_{DD} + 2V_T)$ and $V_{IH} \cong \frac{1}{8}(5V_{DD} - 2V_T)$. For the typical value $V_T = 0.2\,V_{DD}$, this gives

$$NM_L = NM_H = 0.425 \times V_{DD}$$

Hence the available noise margin depends on the power supply voltage level. For V_{DD}=5 V, the noise margin is 2.1 V, and for V_{DD}=3.3 V, the noise margin is 1.4 V.

B.8.5 DYNAMIC OPERATION OF LOGIC GATES

In Figure B.47*a* the node between the two gates is labeled *A*. Because of the way in which transistors are constructed in silicon, N_2 has the effect of contributing to a capacitive load at node *A*. Figure B.43 shows that transistors are constructed by using several layers of different materials. Wherever two types of material meet or overlap inside the transistor, a capacitor may be effectively created. This capacitance is called *parasitic*, or *stray*, capacitance because it results as an undesired side effect of transistor fabrication. In Figure B.47 we are interested in the capacitance that exists at node *A*. A number of parasitic capacitors are attached to this node, somes caused by N_1 and others caused by N_2. One significant parasitic capacitor exists between the input of inverter N_2 and ground. The value of this capacitor depends on the sizes of the transistors in N_2. Each transistor contributes a *gate capacitance*, C_g = W × L × C_{ox}. The parameter C_{ox}, called the *oxide capacitance*, is a constant for the technology being used and has the units fFμm^2. Additional capacitance is caused by the transistors in N_1 and by the metal wiring that is attached to node *A*. It is possible to represent all of the parasitic capacitance by a single equivalent capacitance between node *A* and ground [2]. In Figure B.47*b* this equivalent capacitance is labeled *C*.

The existence of stray capacitance has a negative effect on the speed of operation of logic circuits. Voltage across a capacitor cannot change instantaneously. The time needed to charge or discharge a capacitor depends on the size of the capacitance *C* and on the amount of current through the capacitor. In the circuit of Figure B.47*b*, when the PMOS transistor in N_1 is turned on, the capacitor is charged to V_{DD}; it is discharged when the NMOS transistor is turned on. In each case the current flow I_D through the involved transistor and the value of *C* determine the rate of charging and discharging the capacitor.

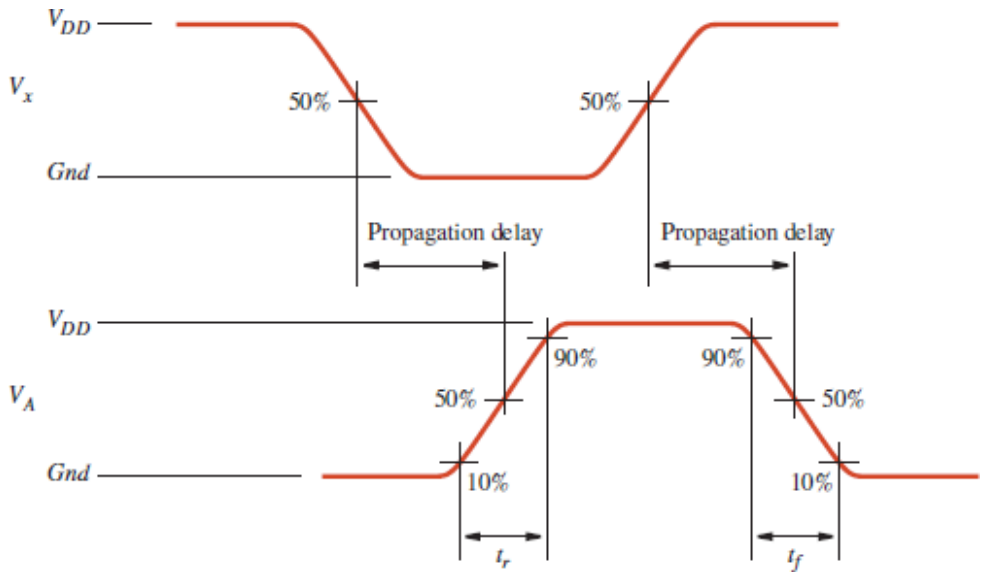


Figure B.48 Voltage waveforms for logic gates.

Chapter 2 introduced the concept of a timing diagram, and Figure 2.10 shows a timing diagram in which waveforms have perfectly-vertical edges in transition from one logic level to the other. In real circuits, waveforms do not have this "ideal" shape, but instead have the appearance of those in Figure B.48. The figure gives a waveform for the input V_x in Figure B.47*b* and shows the resulting waveform at node *A*. We assume that V_x is initially at the voltage level V_{DD} and then makes a transition to 0. Once V_x reaches a sufficiently low voltage, N_1 begins to drive voltage V_A toward V_{DD}. Because of the parasitic capacitance, V_A cannot change instantaneously and a waveform with the shape indicated in the figure results. The time needed for V_A to change from low to high is called the *rise time*, t_r, which is defined as the time elapsed from when V_A is at 10 percent of V_{DD} until it reaches 90 percent of V_{DD}. Figure B.48 also defines the total amount of time needed for the change at V_x to cause a change in V_A. This interval is called the *propagation delay*, often written t_p, of the inverter. It is the time from when V_x reaches 50 percent of V_{DD} until V_A reaches the same level.

After remaining at 0 V for some time, V_x then changes back to V_{DD}, causing N_1 to discharge *C* to *Gnd*. In this case the transition time at node *A* pertains to a change from high to low, which is referred to as the *fall time*, t_f, from 90 percent of V_{DD} to 10 percent of V_{DD}. As indicated in the figure, there is a

corresponding propagation delay for the new change in V_x to affect V_A. In a given logic gate, the relative sizes of the PMOS and NMOS transistors are usually chosen such that t_r and t_f have about the same value.

Equations B.1 and B.2 specify the amount of current flow through an NMOS transistor. Given the value of C in Figure B.47, it is possible to calculate the propagation delay for a change in V_A from high to low. For simplicity, assume that V_x is initially 0 V; hence the PMOS transistor is turned on, and V_A=5 V. Then V_x changes to V_{DD} at time 0, causing the PMOS transistor to turn off and the NMOS to turn on. The propagation delay is defined as the time required to discharge C through the NMOS transistor to the voltage $V_{DD}/2$. When V_x first changes to V_{DD}, V_A=5 V; hence the NMOS transistor will have V_{DS}=V_{DD} and will be in the saturation region. The current I_D is given by Equation B.2. Once V_A drops below V_{DD}-V_T, the NMOS transistor will enter the triode region where I_D is given by Equation B.1. For our purposes, we can approximate the current flow as V_A changes from V_{DD} to $V_{DD}/2$ by finding the average of the values given by Equation B.2 with V_{DS} = V_{DD} and Equation B.1 with V_{DS} = $V_{DD}/2$. Using the basic expression for the time needed to charge a capacitor (see Example B.15), we have

$$t_p = \frac{C\Delta V}{I_D} = \frac{CV_{DD}/2}{I_D}$$

Substituting for the average value of I_D as discussed above, yields [1]

$$t_p \cong \frac{1.7\,C}{k'_n \frac{W}{L} V_{DD}} \qquad \text{[B.4]}$$

This expression specifies that the speed of the circuit depends both on the value of C and on the dimensions of the transistor. The delay can be reduced by making C smaller or by making the ratio W/L larger. The expression shows the propagation time when the output changes from a high level to a low level. The low-to-high propagation time is given by the same expression but using k'_p and W/L of the PMOS transistor.

In logic circuits, L is usually set to the minimum value that is permitted according to the specifications of the fabrication technology used. The value of W is chosen depending on the amount of current flow, hence propagation delay, that is desired. Figure B.49 illustrates two sizes of transistors. Part (*a*)

depicts a minimum-size transistor, which would be used in a circuit wherever capacitive loading is small or where speed of operation is not critical. Figure B.49*b* shows a larger transistor, which has the same length as the transistor in part (*a*) but a larger width. There is a trade-off involved in choosing transistor sizes, because a larger transistor takes more space on a chip than a smaller one. Also, increasing W not only increases the amount of current flow in the transistor but also results in an increase in the parasitic capacitance (recall that the capacitance C_g between the gate terminal and ground is proportional to $W \times L$), which tends to offset some of the expected improvement in performance. In logic circuits large transistors are used where high capacitive loads must be driven and where signal propagation delays must be minimized.

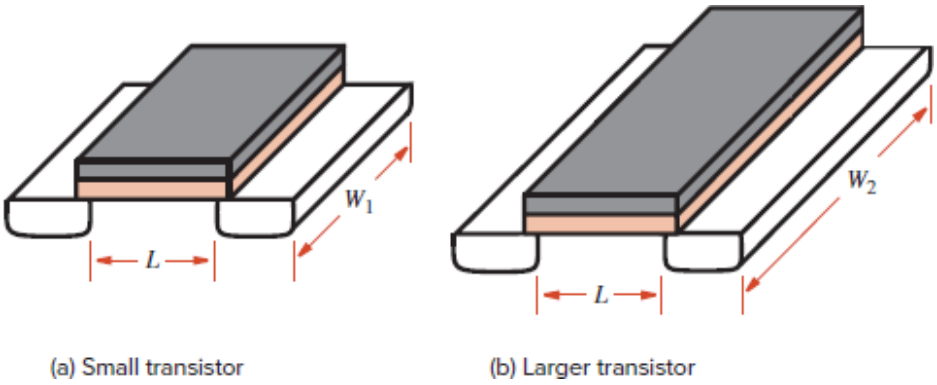


Figure B.49 Transistor sizes.

Example B.7

In the circuit in Figure B.47, assume that C = 70 fF and that W/L = 2.0 µm/0.5 µm. Also, $k'_n = 60\ \mu A/V^2$ and V_{DD}=5 V. Using Equation B.4, the high-to-low propagation delay of the inverter is $t_p \approx 0.1$ ns.

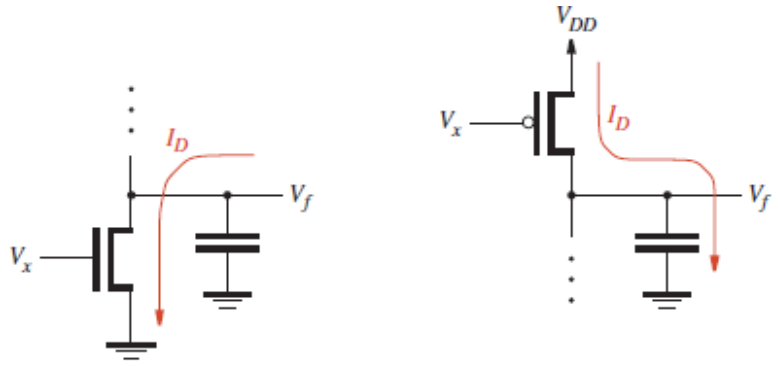
B.8.6 POWER DISSIPATION IN LOGIC GATES

In an electronic circuit it is important to consider the amount of electrical power consumed by the transistors. Integrated circuit technology allows fabrication of billions of transistors on a single chip; hence the amount of power used by an individual transistor must be small. Power dissipation is an important consideration in all applications of logic circuits, but it is crucial in

situations that involve battery-operated equipment, such as portable computers and the like.

Consider again the NMOS inverter in Figure B.45. When $V_x = 0$, no current flows and hence no power is used. But when $V_x = 5$ V, power is consumed because of the current I_{stat}. The power consumed in the steady state is given by $P_S = I_{stat}\ V_{DD}$. In Example B.5 we calculated I_{stat} =0.2 mA. The power consumed is then $P_S = 0.2$ mA × 5 V = 1.0 mW. If we assume that a chip contains, say, the equivalent of 10,000 inverters, then the total power consumption is 10 W! Because of this large power consumption, NMOS-style gates are used only in special-purpose applications, which we discuss in Section B.8.9.

To distinguish between power consumed during steady-state conditions and power consumed when signals are changing, it is customary to define two types of power. *Static power* is dissipated by the current that flows in the steady state, and *dynamic power* is consumed when the current flows because of changes in signal levels. NMOS circuits consume static power as well as dynamic power, while CMOS circuits consume mostly dynamic power. Note that we mentioned in Section B.8.3 that a small amount of current, called *leakage* current, flows through transistors that are turned off. Although this leakage current causes static power dissipation even when using CMOS circuits, we do not consider this effect further in our discussion.



(a) Current flow when input V_x changes from 0 V to 5 V

(b) Current flow when input V_x changes from 5 V to 0 V

Figure B.50 Dynamic current flow in CMOS circuits.

Consider the CMOS inverter presented in Figure B.12*a*. When the input V_x is low, no current flows because the NMOS transistor is off. When V_x is high, the PMOS transistor is off and again no current flows. Hence no current flows

in a CMOS circuit under steady-state conditions. Current does flow in CMOS circuits, however, for a short time when signals change from one voltage level to another.

Figure B.50*a* depicts the following situation. Assume that V_x has been at 0 V for some time; hence V_f = 5 V. Now let V_x change to 5 V. The NMOS transistor turns on, and it pulls V_f toward *Gnd*. Because of the parasitic capacitance *C* at node *f*, voltage V_f does not change instantaneously, and current I_D flows through the NMOS transistor for a short time while the capacitor is being discharged. A similar situation occurs when V_x changes from 5 V to 0, as illustrated in Figure B.50*b*. Here the capacitor *C* initially has 0 volts across it and is then charged to 5 V by the PMOS transistor. Current flows from the power supply through the PMOS transistor while the capacitor is being charged.

The voltage transfer characteristic for the CMOS inverter, shown in Figure B.46, indicates that a range of input voltage V_x exists for which both transistors in the inverter are turned on. Within this voltage range, specifically $V_T < V_x < (V_{DD} - V_T)$, current flows from V_{DD} to *Gnd* through both transistors. This current is often referred to as the *short-circuit current* in the gate. In comparison to the amount of current used to (dis)charge the capacitor *C*, the short-circuit current is negligible in most cases.

The power used by a single CMOS inverter is extremely small. Consider again the situation in Figure B.50*a* when $V_f = V_{DD}$. The amount of energy stored in the capacitor is equal to $C\ V^2_{DD}/2$ (see Example B.12). When the capacitor is discharged to 0 V, this stored energy is dissipated in the NMOS transistor. Similarly, for the situation in Figure B.50*b*, the energy $C\ V^2_{DD}/2$ is dissipated in the PMOS transistor when *C* is charged up to V_{DD}. Thus for each cycle in which the inverter charges and discharges *C*, the amount of energy dissipated is equal to $C\ V^2_{DD}$. Since power is defined as energy used per unit time, the power dissipated in the inverter is the product of the energy used in one discharge/charge cycle times the number of such cycles per second, *f*. Hence the dynamic power consumed is

$$P_D = fCV^2_{DD}$$

In practice, the total amount of dynamic power used in CMOS circuits is significantly lower than the total power needed in other technologies, such as

NMOS. For this reason, virtually all large integrated circuits fabricated today are based on CMOS technology.

Example B.8

For a CMOS inverter, assume that C=70 fF and f = 100 MHz. The dynamic power consumed by the gate is P_D = 175 μW. If we assume that a chip contains the equivalent of 10,000 inverters and that, on average, 20 percent of the gates change values at any given time, then the total amount of dynamic power used in the chip is $P_D = 0.2 \times 10{,}000 \times 175\ \mu\text{W} = 0.35\ \text{W}$.

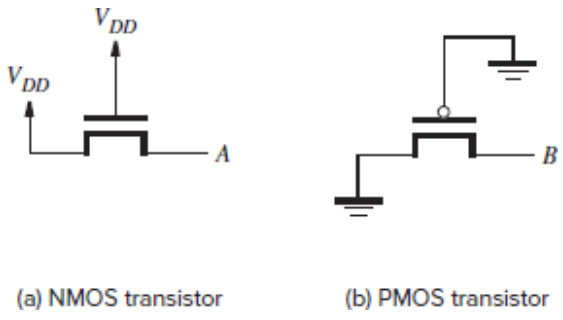
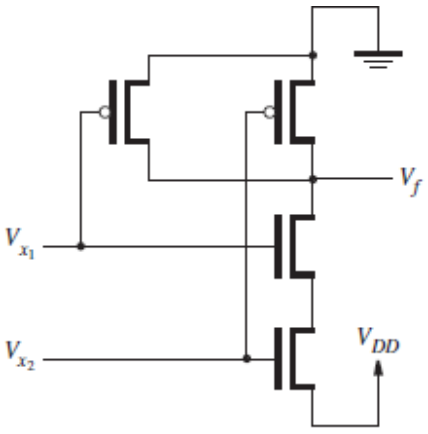


Figure B.51 NMOS and PMOS transistors used in the opposite way from Figure B.4.

B.8.7 Passing 1s and 0s Through Transistor Switches

In Figure B.4 we showed that NMOS transistors are used as pull-down devices and PMOS transistors are used as pull-up devices. We now consider using the transistors in the opposite way, that is, using an NMOS transistor to drive an output high and using a PMOS transistor to drive an output low.

Figure B.51*a* illustrates the case of an NMOS transistor for which both the gate terminal and one side of the switch are driven to V_{DD}. Let us assume initially that both V_G and node A are at 0 V, and we then change V_G to 5 V. Node A is the transistor's source terminal because it has the lowest voltage. Since V_{GS} = V_{DD}, the transistor is turned on and drives node A toward V_{DD}. When the voltage at node A rises, V_{GS} decreases until the point when V_{GS} is no longer greater than V_T. At this point the transistor turns off. Thus in the steady state V_A = V_{DD} - V_T, which means that an NMOS transistor can only partially pass a high voltage signal.



(a) An AND gate circuit

Logic value		Voltage	Logic value
x_1	x_2	V_f	f
0	0	1.5 V	0
0	1	1.5 V	0
1	0	1.5 V	0
1	1	3.5 V	1

(b) Truth table and voltage levels

Figure B.52 A poor implementation of a CMOS AND gate.

A similar situation occurs when a PMOS transistor is used to pass a low voltage level, as depicted in Figure B.51*b*. Here assume that initially both V_G and node *B* are at 5 V. Then we change V_G to 0 V so that the transistor turns on and drives the source node (node *B*) toward 0 V. When node *B* is decreased to V_T, the transistor turns off; hence the steady-state voltage is equal to V_T.

In Section B.1 we said that for an NMOS transistor the substrate (body) terminal is connected to *Gnd* and for a PMOS transistor the substrate is connected to V_{DD}. The voltage between the source and substrate terminals, V_{SB}, which is called the *substrate bias voltage*, is normally equal to 0 V in a logic circuit. But in Figure B.51 both the NMOS and PMOS transistors have $V_{SB} = V_{DD}$. The bias voltage has the effect of increasing the threshold voltage in the transistor V_T by a factor of about 1.5 or higher [2, 1]. This issue is known as the *body effect*.

Consider the logic gate shown in Figure B.52. In this circuit the V_{DD} and *Gnd* connections are reversed from the way in which they were used in previously discussed circuits. When both V_{x_1} and V_{x_2} are high, then V_f is pulled up to the high output voltage, $V_{OH} = V_{DD} - 1.5V_T$. If V_{DD} = 5 V and V_T = 1 V, then V_{OH} = 3.5 V. When either V_{x_1} or V_{x_2} is low, then V_f is pulled down to the low output voltage, $V_{OL} = 1.5\ V_T$, or about 1.5 V. As shown by the truth table in the figure, the circuit represents an AND gate. In comparison to the normal AND gate shown in Figure B.15, the circuit in Figure B.52 appears to be

better because it requires fewer transistors. But a drawback of this circuit is that it offers a lower noise margin because of the poor levels of V_{OH} and V_{OL}.

Another important weakness of the circuit in Figure B.52 is that it causes static power dissipation, unlike a normal CMOS AND gate. Assume that the output of such an AND gate drives the input of a CMOS inverter. When V_f = 3.5 V, the NMOS transistor in the inverter is turned on and the inverter output has a low voltage level. But the PMOS transistor in the inverter is not turned off, because its gate-to-source voltage is -1.5 V, which is larger than V_T. Static current flows from V_{DD} to *Gnd* through the inverter. A similar situation occurs when the AND gate produces the low output V_f = 1.5 V. Here the PMOS transistor in the inverter is turned on, but the NMOS transistor is not turned off. The AND gate implementation in Figure B.52 is not used in practice.

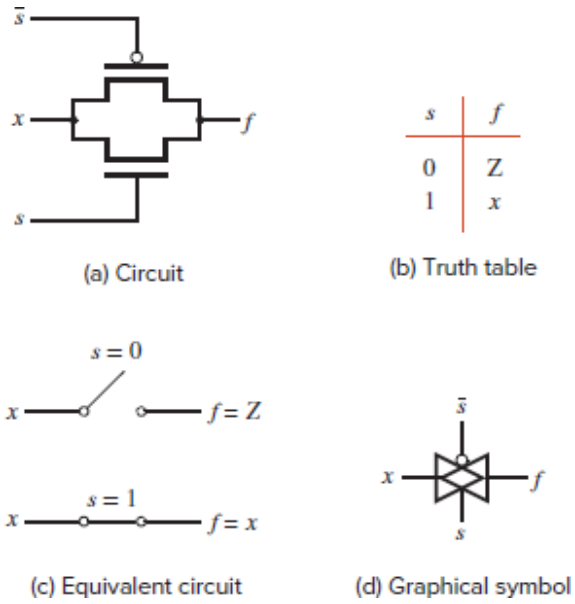


Figure B.53 A transmission gate.

B.8.8 TRANSMISSION GATES

In Section B.8.7 we showed that an NMOS transistor passes 0 well and 1 poorly, while a PMOS transistor passes 1 well and 0 poorly. It is possible to combine an NMOS and a PMOS transistor into a single switch that is capable of driving its output terminal either to a low or high voltage equally well. Figure B.53*a* gives the circuit for a *transmission gate*. As indicated in parts (*b*) and (*c*) of the figure, it acts as a switch that connects *x* to *f*. Switch control is provided by the *select* input *s* and its complement $\bar{s}$. The switch is turned on

by setting $V_s = 5$ V and $V_{\bar{s}} = 0$. When V_x is 0, the NMOS transistor will be turned on (because $V_{GS} = V_s - V_x = 5$ V) and V_f will be 0. On the other hand, when V_x is 5 V, then the PMOS transistor will be on ($V_{GS} = V_{\bar{s}} - V_x = -5$ V) and V_f will be 5 V. A graphical symbol for the transmission gate is given in Figure B.53*d*.

Transmission gates can be used in a variety of applications. The examples below show how they can be used to implement multiplexers, XOR gates, and latch circuits.

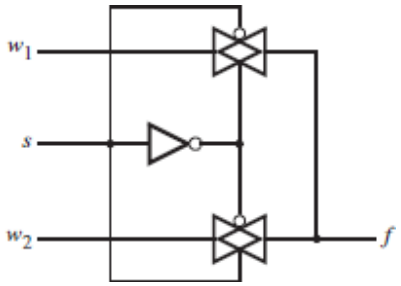


Figure B.54 A 2-to-1 multiplexer built using transmission gates.

Example B.9

We showed a circuit that uses transmission gates to implement a multiplexer in Figure 4.1*d*, which is reproduced in Figure B.54. The select input s is used to choose whether the output f should have the value of input w_1 or w_2. If s=0, then f=w_1; if s=1, then f=w_2.

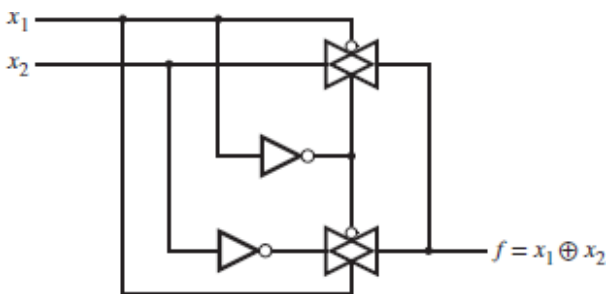


Figure B.55 Implementation of an XOR gate.

Example B.10

The XOR function in sum-of-products form is

$$x_1 \oplus x_2 = \bar{x}_1 x_2 + x_1 \bar{x}_2$$

which leads to a circuit with two NOT gates, 2 two-input AND gates, and a two-input OR gate. We know from Section B.3 that each AND and OR gate requires six transistors, while a NOT gate needs two transistors. Hence 22 transistors are required to implement this circuit in CMOS technology. It is possible to greatly reduce the number of transistors needed by making use of transmission gates. Figure B.55 gives a circuit for an XOR gate that uses the multiplexer structure shown in Figure B.54. The output f is set to the value of x_2 when x_1=0 by the top transmission gate. The bottom transmission gate sets f to $\bar{x}_2$ when x_1=1. This circuit needs only six transistors. We discuss the notion of using multiplexers to implement logic functions in Section 4.1.

Example B.11

Consider the circuit in Figure B.56, which includes two transmission gates. One transmission gate, *TG*1, is used to connect the *Data* input terminal to point *A* in the circuit. The second, *TG*2, is used as a switch in the *feedback loop* that maintains the state of the circuit. The transmission gates are controlled by the *Load* signal. If *Load* = 1, then *TG*1 is on and the point *A* will have the same value as the *Data* input. Since the value presently stored at *Output* may not be the same value as *Data*, the feedback loop is broken by having *TG*2 turned off when *Load* = 1. When *Load* changes to 0, then *TG*1 turns off and *TG*2 turns on. The feedback path is closed and the memory element will retain its state as long as *Load* = 0. This circuit implements a gated D latch. It is equivalent to the circuit with NAND gates shown in Figure 5.7, with the *Load* input in Figure B.56 acting as the *Clk* input.

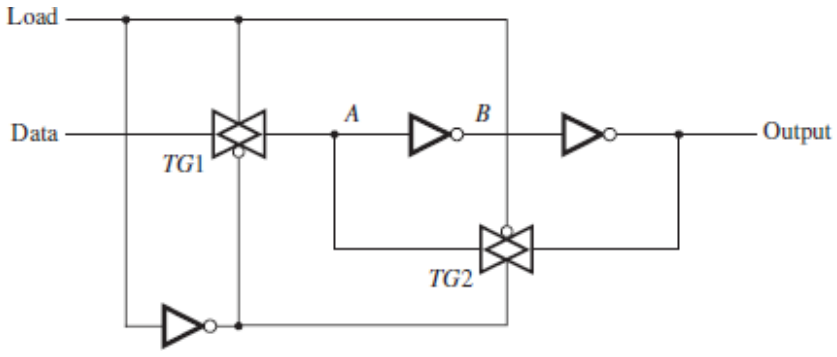


Figure B.56 Using transmission gates to implement a gated D latch.

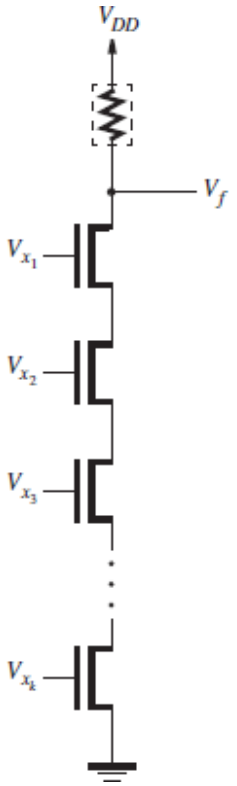


Figure B.57 High fan-in NMOS NAND gate.

B.8.9 Fan-in and Fan-out in Logic Gates

The *fan-in* of a logic gate is defined as the number of inputs to the gate. Depending on how a logic gate is constructed, it may be impractical to increase the number of inputs beyond a small value. For example, consider the NMOS NAND gate in Figure B.57, which has k inputs. We wish to consider the effect of k on the propagation delay t_p through the gate. Assume that all k NMOS transistors have the same width W and length L. Because the transistors are connected in series, we can consider them to be equivalent to one long transistor with length $k \times L$ and width W. Using Equation B.4 (which can be applied to both CMOS and NMOS gates), the propagation delay is given by

$$t_p \cong \frac{1.7\,C}{k'_n \frac{W}{L} V_{DD}} \times k$$

Here C is the equivalent capacitance at the output of the gate, including the parasitic capacitance contributed by each of the k transistors. The performance of the gate can be improved somewhat by increasing W for each NMOS transistor. But this change further increases C and comes at the expense of chip area. Another drawback of the circuit is that each NMOS transistor has the effect of increasing V_{OL}, hence reducing the noise margin. It is practical to build NAND gates in this manner only if the fan-in is small.

As another example of fan-in, Figure B.58 shows an NMOS k-input NOR gate. In this case the k NMOS transistors connected in parallel can be viewed as one large transistor with width $k \times W$ and length L. According to Equation B.4, the propagation delay should be decreased by the factor k. However, the parallel-connected transistors increase the load capacitance C at the gate's output and, more importantly, it is extremely unlikely that all of the transistors would be turned on when V_f is changing from a high to low level. It is thus practical to build high fan-in NOR gates in NMOS technology. We should note, however, that in an NMOS gate the low-to-high propagation delay may be slower than the high-to-low delay as a result of the current-limiting effect of the pull-up device (see Examples B.13 and B.14).

***

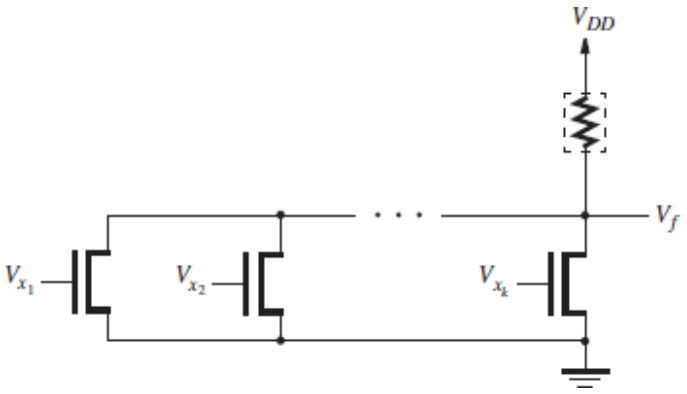


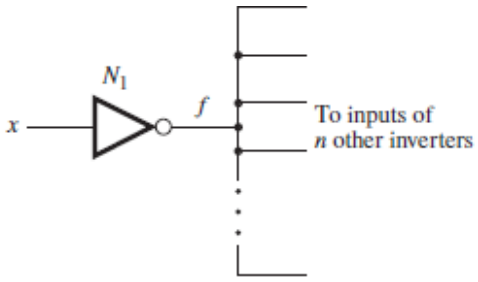
Figure B.58 High fan-in NMOS NOR gate.

High fan-in CMOS logic gates always require either k NMOS or k PMOS transistors in series and are therefore never practical. In CMOS the only reasonable way to construct a high fan-in gate is to use two or more lower fan-in gates. For example, one way to realize a six-input AND gate is as 2 three-input AND gates that connect to a two-input AND gate. It is possible to build a six-input CMOS AND gate using fewer transistors than needed with this approach, but we leave this as an exercise for the reader (see Problem B.4).

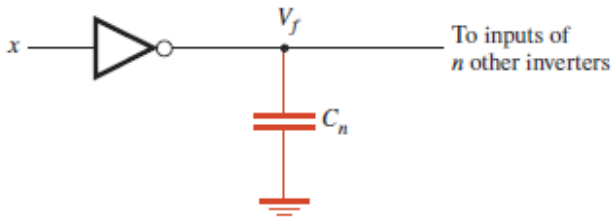
Fan-out

Figure B.48 illustrated timing delays for one NOT gate driving another. In real circuits each logic gate may be required to drive several others. The number of other gates that a specific gate drives is called its *fan-out*. An example of fan-out is depicted in Figure B.59*a*, which shows an inverter N_1 that drives the inputs of n other inverters. Each of the other inverters contributes to the total capacitive loading on node *f*. In part (*b*) of the figure, the n inverters are represented by one large capacitor C_n. For simplicity, assume that each inverter contributes a capacitance C and that $C_n = n \times C$. Equation B.4 shows that the propagation delay increases in direct proportion to n.

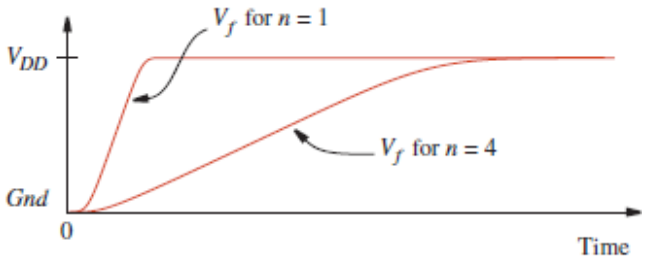
Figure B.59*c* illustrates how n affects the propagation delay. It assumes that a change from logic value 1 to 0 on signal x occurs at time 0. One curve represents the case where n=1, and the other curve corresponds to n=4. Using the parameters from Example B.7, when n=1, we have t_p = 0.1 ns. Then for n=4, $t_p \approx$ 0.4 ns. It is possible to reduce t_p by increasing the W/L ratios of the transistors in N_1.



(a) Inverter that drives n other inverters



(b) Equivalent circuit for timing purposes



(c) Propagation times for different values of n

Figure B.59 The effect of fan-out on propagation delay.

Drivers

In circuits in which a logic gate has to drive a large capacitive load, drivers are often used to improve performance. A *driver* is a logic gate with one input, x, and one output, f, which produces $f = x$. The simplest implementation of a driver uses two inverters, as shown in Figure B.60*a*. Drivers can be created with different amounts of *drive capability*, on the sizes of the transistors (see Figure B.49). In general, because they are used for driving higher-than-normal capacitive loads, drivers have transistors that are larger than those in typical logic gates. The graphical symbol for a non-inverting driver is given in Figure B.60*b*.

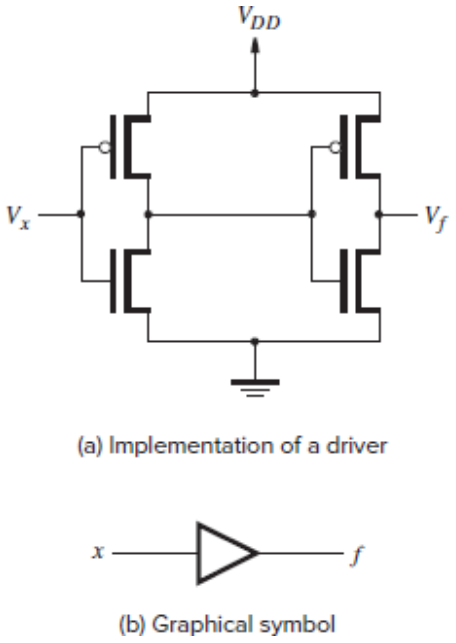


Figure B.60 A non-inverting driver.

Another type of driver is the *inverting driver*. It produces the same output as an inverter, $f = \bar{x}$, but is built with relatively large transistors. The graphical symbol for the inverting driver is the same as for the NOT gate; an inverting driver is just a NOT gate that is capable of driving large capacitive loads. In Figure B.59 for large values of n an inverting driver could be used for the inverter labeled N_1.

In addition to their use for improving the speed performance of circuits, drivers are also used when high current flow is needed to drive external devices. Drivers can handle relatively large amounts of current flow because they are built with large transistors. A common example of this use of drivers is to control a light-emitting diode (LED). We describe an example of this application of drivers in Section 5.14.

In general, fan-out, capacitive loading, and current flow are important issues that the designer of a digital circuit must consider carefully. In practice, the decision as to whether or not drivers are needed in a circuit is made with the aid of CAD tools.

B.8.10 TRI-STATE DRIVERS

As we showed in Figure 7.1, a tri-state driver has one input, *w*, one output, *f*, and a control input, called *enable*, *e*. When *e*=0, the driver is completely disconnected from the output *f*, but when *e*=1, the driver provides the value of

w on the output *f*. Figure B.61 shows a possible implementation of a tri-state driver using NOT gates and a transmission gate.

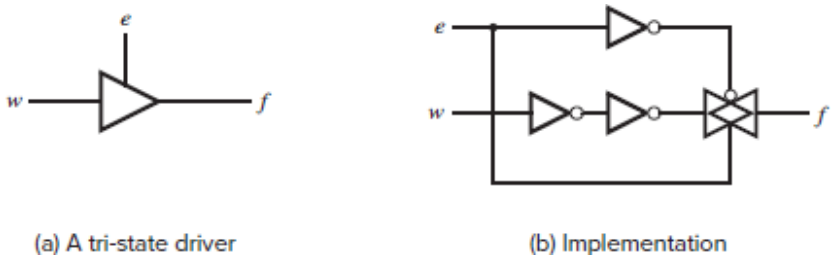


Figure B.61 Tri-state driver.

Figure B.62 depicts several types of tri-state drivers. The driver in part (*b*) has the same behavior as the driver in part (*a*), except that when *e*=1, it produces $f = \overline{w}$. Part (*c*) of the figure gives a tri-state driver for which the enable signal has the opposite behavior; that is, when *e*=0, *f*=*w*, and when *e*=1, *f*=*Z*. The term often used to describe this type of behavior is to say that the enable is *active low*. The driver in Figure B.62*d* also features an active-low enable, and it produces $f = \overline{w}$ when *e*=0.

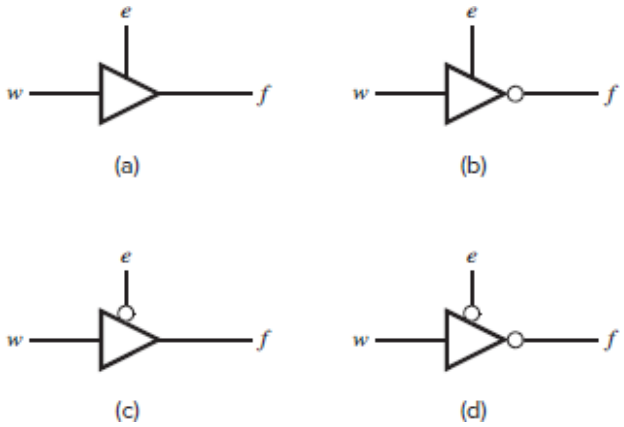


Figure B.62 Four types of tri-state drivers.

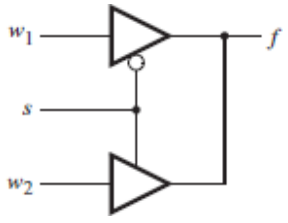


Figure B.63 An application of tri-state drivers.

As a small example of how tri-state drivers can be used, consider the circuit in Figure B.63. In this circuit the output f is equal to either w_1 or w_2, depending on the value of s. When s=0, f=w_1, and when s=1, f=w_2. Thus, this circuit implements a 2-to-1 multiplexer.

In the circuit of Figure B.63, the outputs of the tri-state drivers are wired together. This connection is possible because the control input s is connected so that one of the two drivers is guaranteed to be in the high-impedance state. The w_1 driver is active only when s = 0, and the w_2 driver is active only when s = 1. It would be disastrous to allow both drivers to be active at the same time. Doing so would create a short circuit between V_{DD} and *Gnd* as soon as the two drivers produce different values. For example, assume that w_1 = 1 and w_2 = 0. The w_1 driver produces the output V_{DD}, and the w_2 driver produces *Gnd*. A short circuit is formed between V_{DD} and *Gnd*, through the transistors in the tri-state drivers. The amount of current that flows through such a short circuit is usually sufficient to destroy the circuit.

The kind of *wired* connection used for the tri-state drivers is not possible with ordinary logic gates, because their outputs are always active; hence a short circuit would occur. As we already know, for normal logic circuits the equivalent result of the wired connection is achieved by using an OR gate to combine signals, as is done in the sum-of-products form.

B.9 Static Random Access Memory (SRAM)

In Chapter 5 we introduced several types of circuits that can be used to store data. Assume that we need to store a large number, m, of data items, each of which consists of n bits. One possibility is to use an n-bit register that consists of n D flip-flops for each data item. We would need to design circuitry to control access to each register, both for loading (writing) data into it and for reading data out.

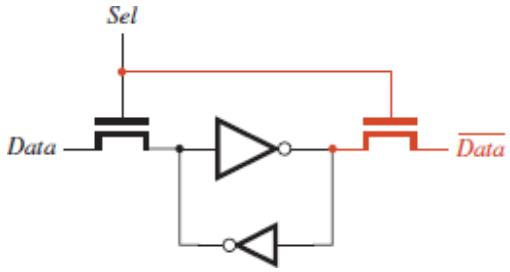


Figure B.64 An SRAM cell.

When *m* is large, it is awkward to use individual registers to store the data. A better approach is to make use of a *static random access memory* (SRAM) block. An SRAM block is a two-dimensional array of SRAM cells, where each cell can store one bit of information. If we need to store *m* items with *n* bits each, we can use an array of $m \times n$ SRAM cells. The dimensions of the SRAM array are called its *aspect ratio*.

An SRAM block may contain a large number of SRAM cells, so it is important for each cell to take as little space on an integrated circuit chip as possible. For this reason, the storage cell should use as few transistors as possible. One popular storage cell used in practice is depicted in Figure B.64. It operates as follows. To store data into the cell, the *Sel* input is set to 1, and the data value to be stored is placed on the *Data* input. The SRAM cell may include a separate input for the complement of the data, indicated by the transistor shown in red in the figure. For simplicity we assume that this transistor is not included in the cell. After waiting long enough for the data to propagate through the feedback path formed by the two NOT gates, *Sel* is changed to 0. The stored data then remains in the feedback loop indefinitely. A possible problem is that when *Sel* = 1, the value of *Data* may not be the same as the value being driven by the small NOT gate in the feedback path. Hence the transistor controlled by *Sel* may attempt to drive the stored data to one logic value while the output of the small NOT gate has the opposite logic value. To resolve this problem, the NOT gate in the feedback path is built using small (weak) transistors, so that its output can be overridden with new data.

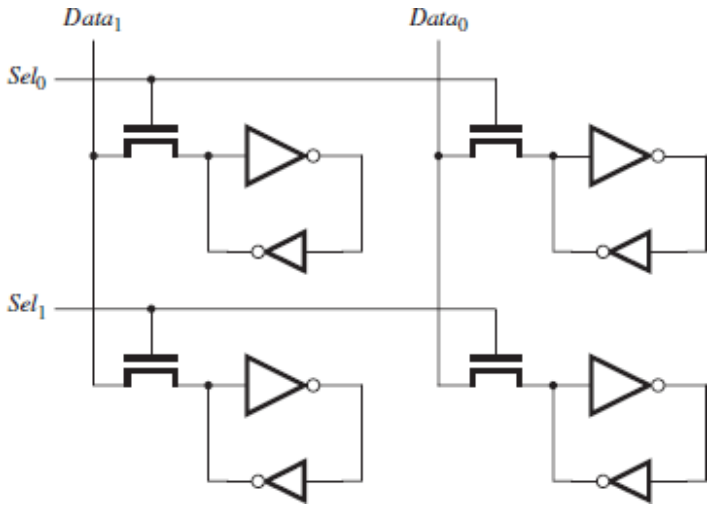


Figure B.65 A 2 × 2 array of SRAM cells.

To read data stored in the cell, we simply set *Sel* to 1. In this case the *Data* node would not be driven to any value by external circuitry, so that the SRAM cell can place the stored data on this node. The *Data* signal is passed through a driver, not shown in the figure, and provided as an output of the SRAM block.

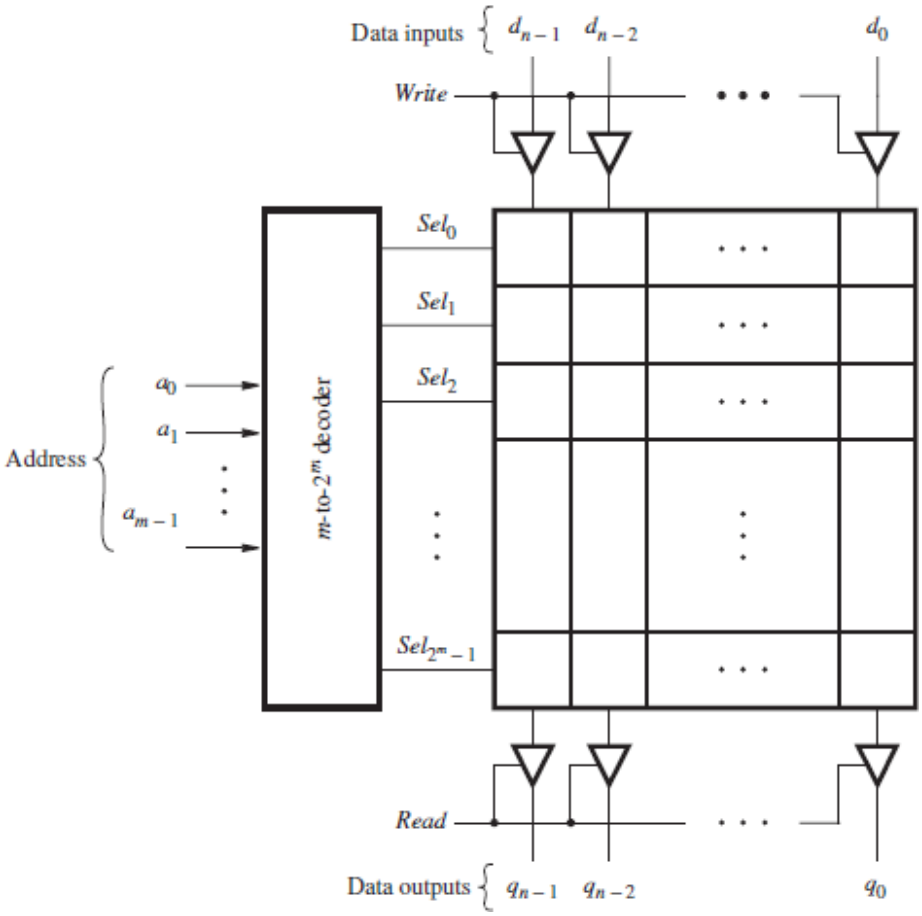


Figure B.66 A $2^m \times n$ SRAM block.

An SRAM block contains an array of SRAM cells. Figure B.65 shows an array with two rows of two cells each. In each column of the array, the *Data* nodes of the cells are connected together. Each row, i, has a separate select input, Sel_i, that is used to read or write the contents of the cells in that row. Larger arrays are formed by connecting more cells to Sel_i in each row and by adding more rows. The SRAM block must also contain circuitry that controls access to each row in the array. Figure B.66 depicts a $2^m \times n$ array of the type in Figure B.65, with a decoder that drives the *Sel* inputs in each row of the array. The inputs to the decoder are called *Address* inputs. This term derives from the notion that the location of a row in the array can be thought of as the "address" of the row. The decoder has m *Address* inputs and produces 2^m select outputs. If the *Write* control input is 1, then the tri-state drivers at the top of the figure are enabled and the data bits on the inputs d_{n-1}, …, d_0 are stored in the cells of the row selected by the *Address* inputs. If the *Read* control input is 1, then the tri-state drivers at the bottom of Figure B.66 are

enabled and the data stored in the row selected by the *Address* inputs appears on the q_{n-1}, …, q_0 outputs. In many practical applications the data inputs and data outputs are connected together. Thus the *Write* and *Read* inputs must never have the value 1 at the same time.

The design of memory blocks has been the subject of intensive research and development. We have described only the basic operation of one type of memory block. The reader can refer to books on computer organization for more information [14, 15].

B.9.1 SRAM Blocks in PLDs

Most FPGA chips contain SRAM blocks that can be used as part of logic circuits implemented in the chips. One popular chip has a number of SRAM blocks, each of which contains 4096 SRAM cells. The SRAM blocks can be configured to provide different aspect ratios, depending on the needs of the design being implemented. Aspect ratios from 512 × 8 to 4096 × 1 can be realized using a single SRAM block, and multiple blocks can be combined to form larger memory arrays. To include SRAM blocks in a circuit, designers use prebuilt modules that are provided in a library as part of the CAD tools, or they write VHDL code from which synthesis tools can infer memory blocks.

B.10 Implementation Details for SPLDs, CPLDs, and FPGAs

We introduced PLDs in Section B.6. In the chip diagrams shown in that section, the programmable switches are represented using the symbol X. We now show how these switches are implemented using transistors.

In commercial SPLDs two main technologies are used to manufacture the programmable switches. The oldest technology is based on using metal-alloy fuses as programmable links. In this technology each pair of horizontal and vertical wires that cross is connected by a small metal fuse. When the chip is programmed, for every connection that is not wanted in the circuit being implemented, the associated fuse is melted. The programming process is not reversible, because the melted fuses are destroyed. We will not elaborate on this technology, because it has mostly been replaced by another method.

In currently produced PLAs and PALs, programmable switches are implemented using a special type of *programmable transistor*. Because CPLDs comprise PAL-like blocks, the technology used in SPLDs is also applicable to CPLDs. We will illustrate the main ideas by first describing PLAs. For a PLA to be useful for implementing a wide range of logic functions, it should support both functions of only a few variables and functions of many variables. In Section B.8.9 we discussed the issue of fan-in of logic gates. We showed that when the fan-in is high, the best type of gate to use is the NMOS NOR gate. Hence PLAs are usually based on this type of gate.

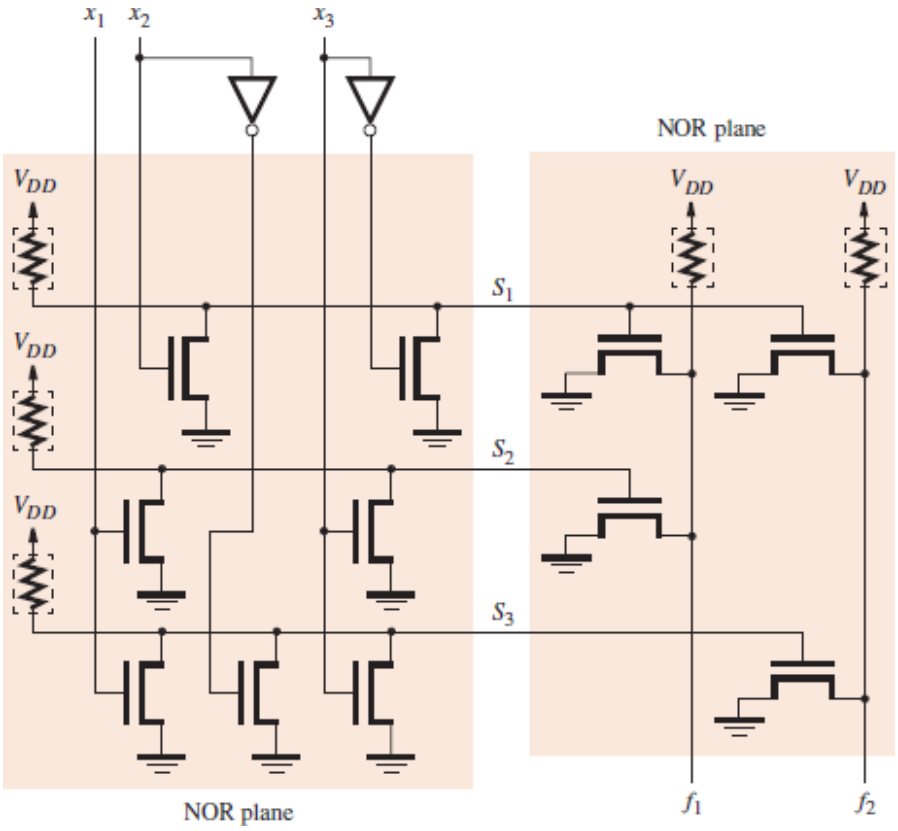


Figure B.67 An example of a NOR-NOR PLA.

As a small example of PLA implementation, consider the circuit in Figure B.67. The horizontal wire labeled S_1 is the output of an NMOS NOR gate with the inputs x_2 and $\bar{x}_3$. Thus $S_1 = \overline{x_2 + \bar{x}_3}$. Similarly, S_2 and S_3 are the outputs of NOR gates that produce $S_2 = \overline{x_1 + x_3}$ and $S_3 = \overline{x_1 + \bar{x}_2 + x_3}$. The three NOR gates that produce S_1, S_2, and S_3 are arranged in a regular structure that is efficient to create on an integrated circuit. This structure is called a *NOR plane*. The NOR

plane is extended to larger sizes by adding columns for additional inputs and adding rows for more NOR gates.

The signals S_1, S_2, and S_3 serve as inputs to a second NOR plane. This NOR plane is turned 90 degrees clockwise with respect to the first NOR plane to make the diagram easier to draw. The NOR gate that produces the output f_1 has the inputs S_1 and S_2. Thus

$$f_1 = \overline{S_1 + S_2} = \overline{\overline{(x_2 + \bar{x}_3)} + \overline{(x_1 + x_3)}}$$

Using DeMorgan's theorem, this expression is equivalent to the product-of-sums expression

$$f_1 = \bar{S}_1 \bar{S}_2 = (x_2 + \bar{x}_3)(x_1 + x_3)$$

Similarly, the NOR gate with output f_2 has inputs S_1 and S_3. Therefore,

$$f_2 = \overline{S_1 + S_3} = \overline{\overline{(x_2 + \bar{x}_3)} + \overline{(x_1 + \bar{x}_2 + x_3)}}$$

which is equivalent to

$$f_2 = \bar{S}_1 \bar{S}_3 = (x_2 + \bar{x}_3)(x_1 + \bar{x}_2 + x_3)$$

The style of PLA illustrated in Figure B.67 is called a *NOR-NOR PLA*. Alternative implementations also exist, but because of its simplicity, the NOR-NOR style is the most popular choice. The reader should note that the PLA in Figure B.67 is not programmable—with the transistors connected as shown, it realizes only the two specific logic functions f_1 and f_2. But the NOR-NOR structure can be used in a programmable version of the PLA, as explained below.

Strictly speaking, the term PLA should be used only for the fixed type of PLA depicted in Figure B.67. The proper technical term for a programmable type of PLA is *field-programmable logic array (FPLA)*. However, it is common usage to omit the *F*. Figure B.68*a* shows a programmable version of a NOR plane. It has n inputs, x_1, …, x_n, and k outputs, S_1, …, S_k. At each crossing point of a horizontal and vertical wire there exists a programmable switch. The switch comprises two transistors connected in series, an NMOS transistor and an *electrically erasable programmable read-only memory (EEPROM)* transistor.

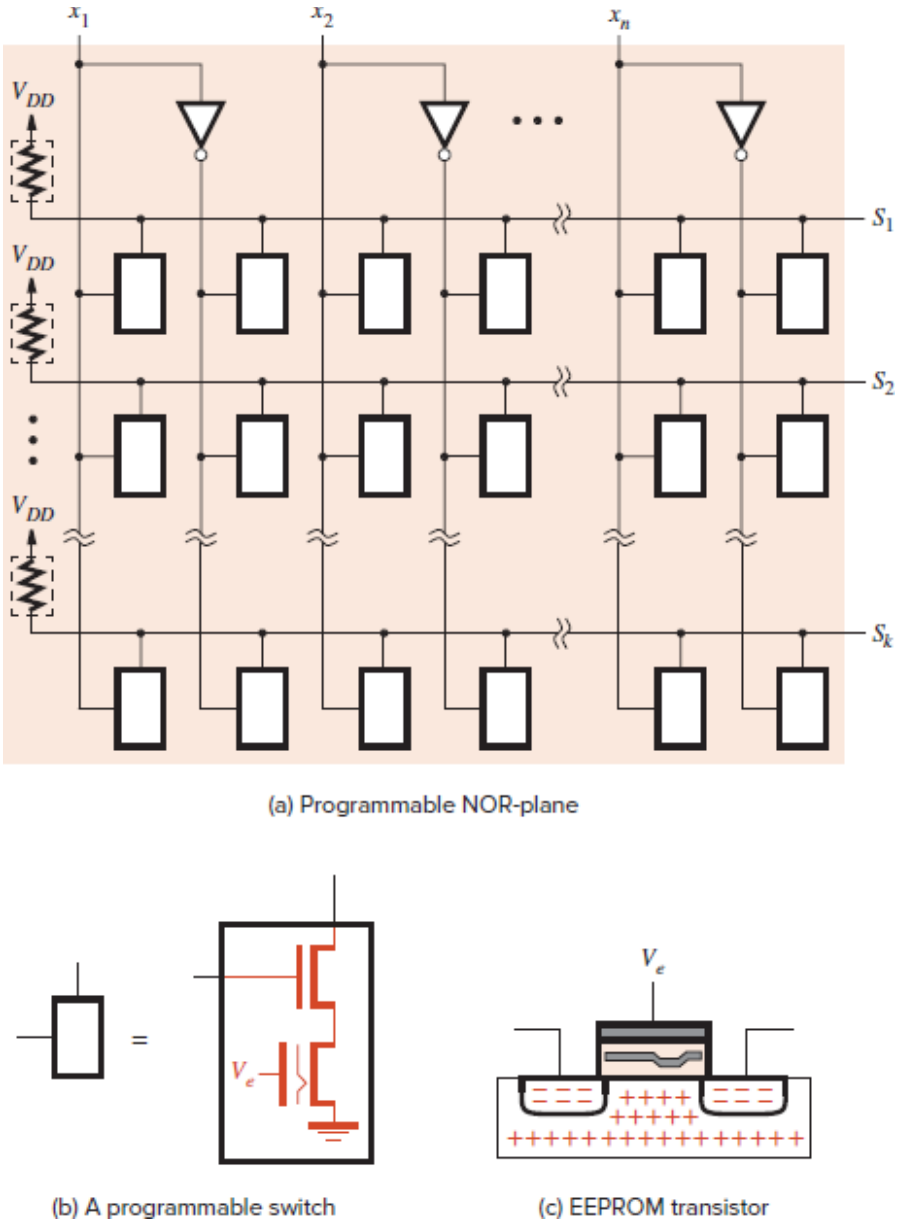


Figure B.68 Using EEPROM transistors to create a programmable NOR plane.

The programmable switch is based on the behavior of the EEPROM transistor. Electronics textbooks, such as [1, 2], give detailed explanations of how EEPROM transistors operate. Here we will provide only a brief description. A programmable switch is depicted in Figure B.68*b*, and the structure of the EEPROM transistor is given in Figure B.68*c*. The EEPROM transistor has the same general appearance as the NMOS transistor (see Figure B.43) with one major difference. The EEPROM transistor has two gates: the normal gate that an NMOS transistor has and a second *floating gate*. The floating gate is so named because it is surrounded by insulating glass and is

not connected to any part of the transistor. When the transistor is in the original unprogrammed state, the floating gate has no effect on the transistor's operation and it works as a normal NMOS transistor. During normal use of the PLA, the voltage on the floating gate V_e is set to V_{DD} by circuitry not shown in the figure, and the EEPROM transistor is turned on.

Programming of the EEPROM transistor is accomplished by turning on the transistor with a higher-than-normal voltage level (typically, V_e = 12 V), which causes a large amount of current to flow through the transistor's channel. Figure B.68*c* shows that a part of the floating gate extends downward so that it is very close to the top surface of the channel. A high current flowing through the channel causes an effect, known as *Fowler-Nordheim tunneling*, in which some of the electrons in the channel "tunnel" through the insulating glass at its thinnest point and become trapped under the floating gate. After the programming process is completed, the trapped electrons repel other electrons from entering the channel. When the voltage $V_e = V_{DD}$ is applied to the EEPROM transistor, which would normally cause it to turn on, the trapped electrons keep the transistor turned off. Hence in the NOR plane in Figure B.68*a*, programming is used to "disconnect" inputs from the NOR gates. For the inputs that should be connected to each NOR gate, the corresponding EEPROM transistors are left in the unprogrammed state.

Once an EEPROM transistor is programmed, it retains the programmed state permanently. However, the programming process can be reversed. This step is called *erasing*, and it is done by using voltages that are of the opposite polarity to those used for programming. In this case, the applied voltage causes the electrons that are trapped under the floating gate to tunnel back to the channel. The EEPROM transistor returns to its original state and again acts like a normal NMOS transistor.

For completeness, we should also mention another technology that is similar to EEPROM, called *erasable PROM (EPROM)*. This type of transistor, which was actually created as the predecessor of EEPROM, is programmed in a similar fashion to EEPROM. However, erasing is done differently: to erase an EPROM transistor, it must be exposed to light energy of specific wavelengths. To facilitate this process, chips based on EPROM technology are housed in packages with a clear glass window through which the chip is visible. To erase a chip, it is placed under an *ultraviolet light* source for several minutes. Because erasure of EPROM transistors is more awkward than the electrical process used to erase EEPROM transistors, EPROM technology has essentially been replaced by EEPROM technology in practice.

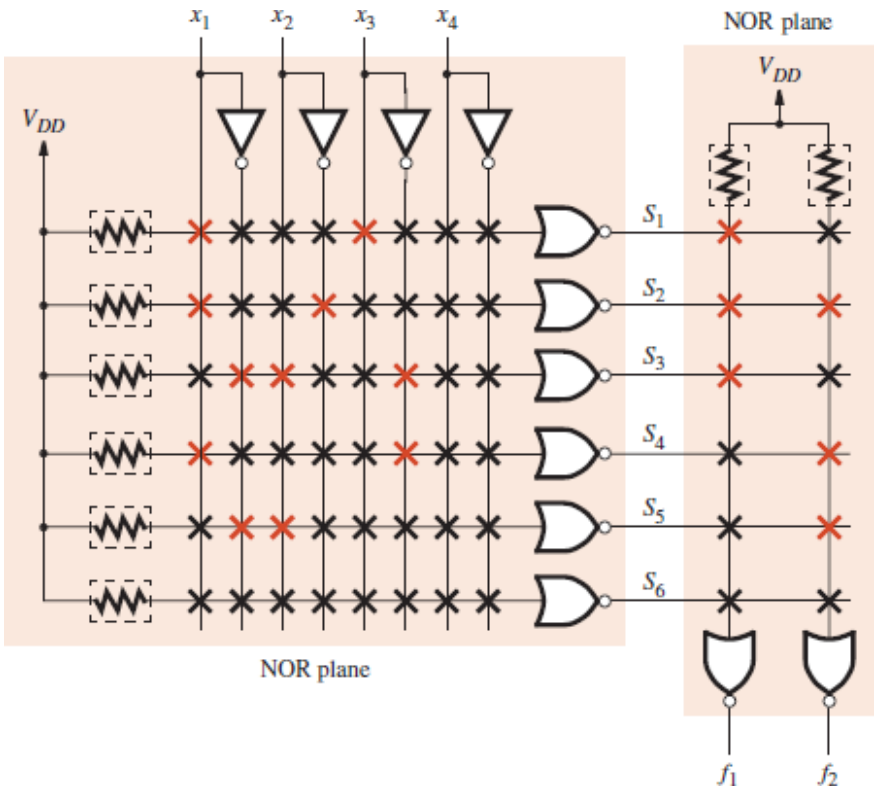


Figure B.69 Programmable version of the NOR-NOR PLA.

A complete NOR-NOR PLA using EEPROM technology, with four inputs, six sum terms in the first NOR plane, and two outputs, is depicted in Figure B.69. Each programmable switch that is programmed to the off state is shown as X in black, and each switch that is left unprogrammed is shown in red. With the programming states shown in the figure, the PLA realizes the logic functions $f_1 = (x_1 + x_3)(x_1 + \bar{x}_2)(\bar{x}_1 + x_2 + \bar{x}_3)$ and $f_2 = (x_1 + \bar{x}_3)(\bar{x}_1 + x_2)(x_1 + \bar{x}_2)$.

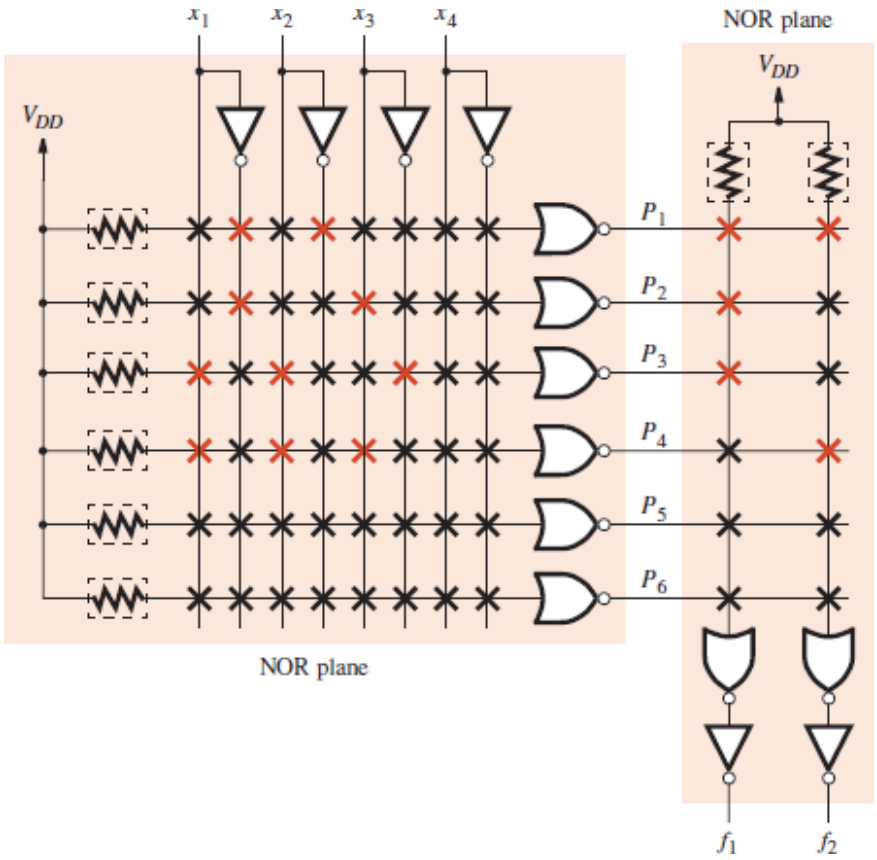


Figure B.70 A NOR-NOR PLA used for sum-of-products.

Rather than implementing logic functions in product-of-sums form, a PLA can also be used to realize the sum-of-products form. For sum-of-products we need to implement AND gates in the first NOR plane of the PLA. If we first complement the inputs to the NOR plane, then according to DeMorgan's theorem, this is equivalent to creating an AND plane. We can generate the complements at no cost in the PLA because each input is already provided in both true and complemented forms. An example that illustrates implementation of the sum-of-products form is given in Figure B.70. The outputs from the first NOR plane are labeled P_1, …, P_6 to reflect our interpretation of them as product terms. The signal P_1 is programmed to realize $\overline{\bar{x}_1 + \bar{x}_2} = x_1 x_2$. Similarly, $P_2 = x_1 \bar{x}_3$, $P_3 = \bar{x}_1 \bar{x}_2 x_3$, and $P_4 = \bar{x}_1 \bar{x}_2 \bar{x}_3$. Having generated the desired product terms, we now need to OR them. This operation can be accomplished by complementing the outputs of the second NOR plane. Figure B.70 includes NOT gates for this purpose. The states indicated for the programmable switches in the OR plane (the second NOR plane) in the figure

yield the following outputs: $f_1 = P_1 + P_2 + P_3 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$, and $f_2 = P_1 + P_4 = x_1x_2 + \bar{x}_1\bar{x}_2\bar{x}_3$.

The concepts described above for PLAs can also be used in PALs. Figure B.71 shows a PAL with four inputs and two outputs. Let us assume that the first NOR plane is programmed to realize product terms in the manner described above. Notice in the figure that the product terms are hardwired in groups of three to OR gates that produce the outputs of the PAL. As we illustrated in Figure B.29, the PAL may also contain extra circuitry between the OR gates and the output pins, which is not shown in Figure B.71. The PAL is programmed to realize the same logic functions, f_1 and f_2, that were generated in the PLA in Figure B.70. Observe that the product term x_1 x_2 is implemented twice in the PAL, on both P_1 and P_4. Duplication is necessary because in a PAL product terms cannot be shared by multiple outputs, as they can be in a PLA. Another detail to observe in Figure B.71 is that although the function f_2 requires only two product terms, each OR gate is hardwired to three product terms. The extra product term P_6 must be set to logic value 0, so that it has no effect. This is accomplished by programming P_6 so that it produces the product of an input and that input's complement, which always results in 0. In the figure, $P_6 = x_1\bar{x}_1 = 0$, but any other input could also be used for this purpose.

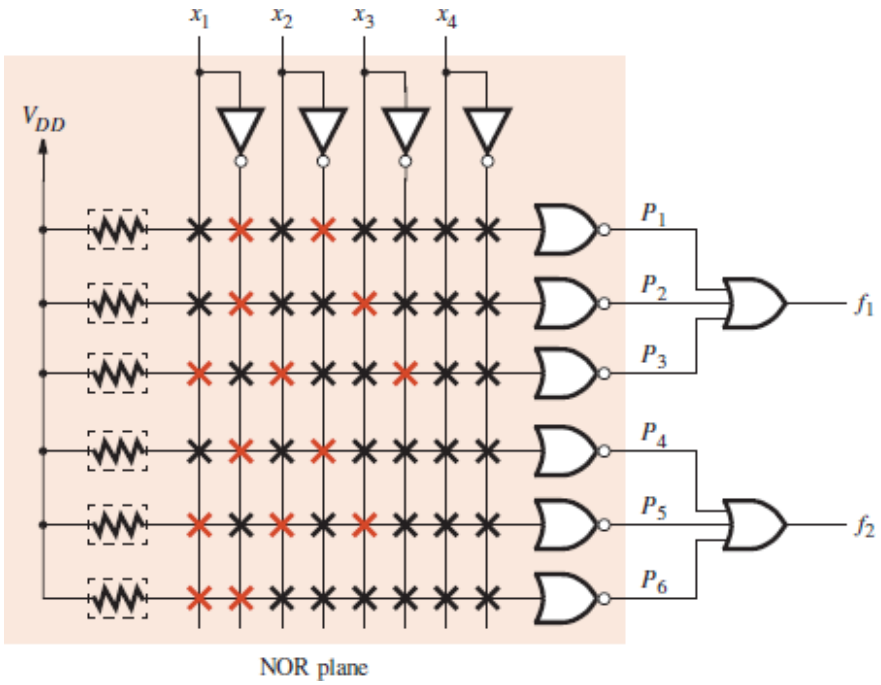


Figure B.71 PAL programmed to implement the functions in Figure B.70.

The PAL-like blocks contained in CPLDs are usually implemented using the techniques discussed in this section. In a typical CPLD, the AND plane is built using NMOS NOR gates, with appropriate complementing of the inputs. The OR plane is hardwired as it is in a PAL, rather than being fully programmable as in a PLA. However, some flexibility exists in the number of product terms that feed each OR gate. This flexibility is accomplished by using a programmable circuit that can allocate the product terms to whichever OR gates the user desires.

Example B.12

In Section B.9 we described static random access memory. This type of memory is useful when it is necessary to support both reading and writing of data. In many situations it is also useful to be able to write data into a memory and then use this stored data permanently without further modification. A read-only memory (ROM) is suitable for such applications. In a ROM the storage cells are built using EEPROM (or EPROM) transistors, arranged in a structure similar to that shown in Figure B.68. Each data bit in the ROM normally provides a 1 if its corresponding EEPROM cell is programmed to be “off", but this data bit is pulled down to 0 if the EEPROM cell is programmed to be “ON".

Figure B.72 shows an example of a ROM block. The storage cells are arranged in 2^m rows with n cells per row. Thus each row stores n bits of information. The location of each row in the ROM is identified by its *address*. In the figure the row at the top has address 0, and the row at the bottom has address 2^m-1. The information stored in the rows can be accessed by asserting the select lines, Sel_0 to Sel_{2^m-1}. As shown in the figure, a decoder with m inputs and 2^m outputs is used to generate the signals on the select lines. Figure B.72 shows that each data line has an associated tri-state driver that is enabled by the ROM input named *Read*. To access, or *read*, data from the ROM, the address of the desired row is placed on the address lines and *Read* is set to 1.

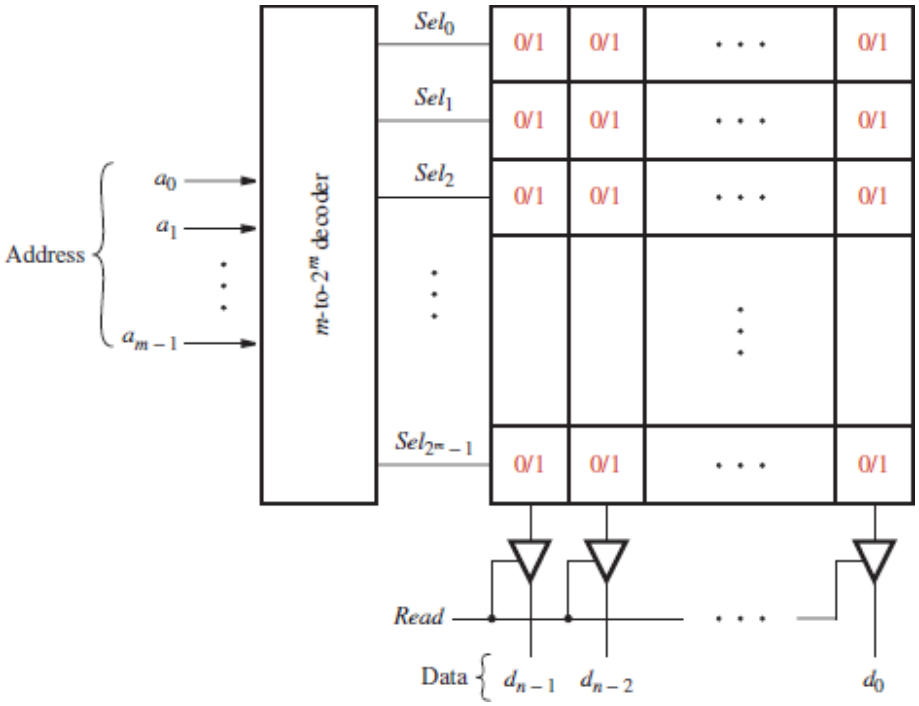


Figure B.72 A $2^m \times n$ read-only memory (ROM) block.

B.10.1 IMPLEMENTATION IN FPGAS

FPGAs do not use EEPROM technology to implement the programmable switches. Instead, the programming information is stored in SRAM cells. An SRAM cell is used for each truth-table value stored in a LUT. SRAM cells are also used to configure the interconnection wires in an FPGA.

Figure B.73 depicts a small section of the FPGA from Figure B.39. The logic element shown produces the output f_1, which is driven onto the horizontal wire drawn in red. This wire can be connected to some of the vertical wires that it crosses, using programmable switches. Each switch is implemented using an NMOS transistor, with its gate terminal controlled by an SRAM cell. Such a switch is known as a *pass-transistor switch*. If a 0 is stored in an SRAM cell, then the associated NMOS transistor is turned off. But if a 1 is stored in the SRAM cell, as shown for the switch drawn in red, then the NMOS transistor is turned on. This switch forms a connection between the two wires attached to its source and drain terminals. The number of switches that are provided in the FPGA depends on the specific chip architecture.

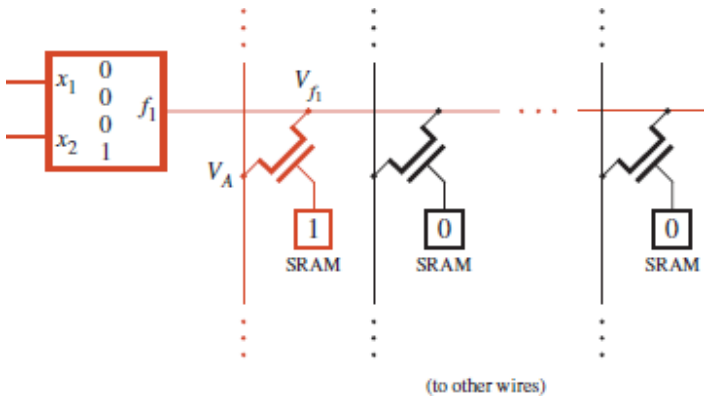


Figure B.73 Pass-transistor switches in FPGAs.

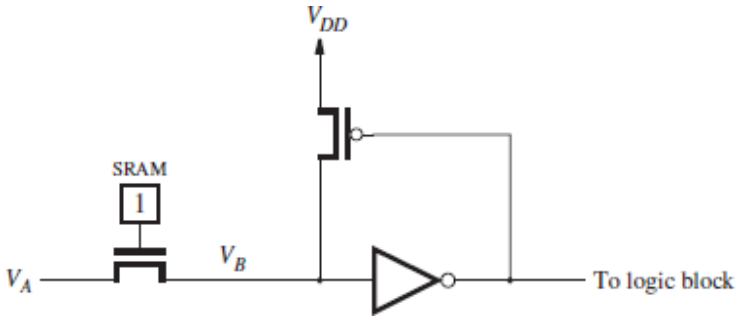


Figure B.74 Restoring a high voltage level.

In Section B.8.7 we showed that an NMOS transistor can only partially pass a high logic value. Hence in Figure B.73 if V_{f_1} is a high voltage level, then V_A is only partially high. Using the values from Section B.8.7, if V_{f_1}= 5 V, then V_A=3.5 V. As we explained in Section B.8.7, this degraded voltage level has the result of causing static power to be consumed (see Example B.19). One solution to this problem [1] is illustrated in Figure B.74. We assume that the signal V_A passes through another pass-transistor switch before reaching its destination at another logic element. The signal V_B has the same value as V_A because the threshold voltage drop occurs only when passing through the first pass-transistor switch. To restore the level of V_B, it is driven with an inverter. A PMOS transistor is connected between the input of this inverter and V_{DD}, and that transistor is controlled by the inverter's output. The PMOS transistor has no effect on the inverter's output voltage level when V_B = 0 V. But when V_B = 3.5 V, then the inverter output is low, which turns on the PMOS transistor. This transistor quickly restores V_B to the proper level of V_{DD}, thus preventing current from flowing in the steady state. Instead of using this pull-

up transistor solution, another possible approach is to alter the threshold voltage of the PMOS transistor (during the integrated circuit manufacturing process) in the inverter in Figure B.74, such that the magnitude of its threshold voltage is large enough to keep the transistor turned off when V_B = 3.5 V. In commercial FPGAs both of these solutions are used in different chips.

An alternative to using a single NMOS transistor as a switch is to use a transmission gate, described in Section B.8.8. While this approach solves the voltage-level problem, it has two drawbacks. First, having both an NMOS and PMOS transistor in the switch increases the capacitive loading on the interconnection wires, which increases the propagation delays and power consumption. Second, the transmission gate takes more chip area than does a single NMOS transistor. For these reasons, commercial FPGA chips do not usually use transmission-gate switches.

B.11 Concluding Remarks

We have described the most important concepts that are needed to understand how logic gates are built using transistors. Our discussions of transistor fabrication, voltage levels, propagation delays, power dissipation, and the like are meant to give the reader an appreciation of the practical issues that have to be considered when designing and using logic circuits.

We have introduced several types of integrated circuit chips. Each type of chip is appropriate for different applications. The standard chips, such as the 7400 series, contain only a few simple gates and are rarely used today. Exceptions to this are the driver chips, which are employed when large amounts of current are needed. The various types of PLDs are widely used in many types of applications. Simple PLDs, like PLAs and PALs, are appropriate for implementation of small logic circuits. The SPLDs offer low cost and high speed. CPLDs can be used for the same applications as SPLDs, but CPLDs are also well suited for implementation of larger circuits, up to about 10,000 to 20,000 gates. Many of the applications that can be targeted to CPLDs can alternatively be realized with FPGAs. Which of these two types of chips are used in a specific design situation depends on many factors. Following the trend of putting as much circuitry as possible into a single chip,

FPGAs are more widely used than other PLDs. Most digital designs created in the industry today contain some type of FPGA.

The gate-array, standard-cell, and custom-chip technologies are used in cases where PLDs are not appropriate. Typical applications are those that entail very large circuits, require extremely high speed-of-operation, need low power consumption, and where the designed product is expected to sell in large volume.

B.12 Examples of Solved Problems

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

Example B.13

Problem: We introduced standard cell technology in Section B.7. In this technology, circuits are built by interconnecting building-block cells that implement simple functions, like basic logic gates. A commonly used type of standard cell are the and-or-invert (AOI) cells, which can be efficiently built as CMOS complex gates. Consider the AOI cell shown in Figure B.75. This cell implements the function $f = \overline{x_1x_2 + x_3x_4 + x_5}$. Derive the CMOS complex gate that implements this cell.

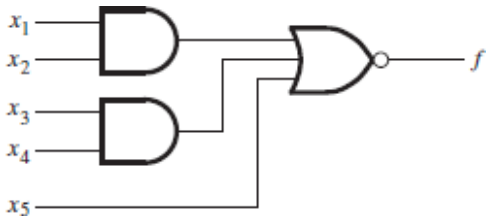


Figure B.75 The AOI cell for Example B.13.

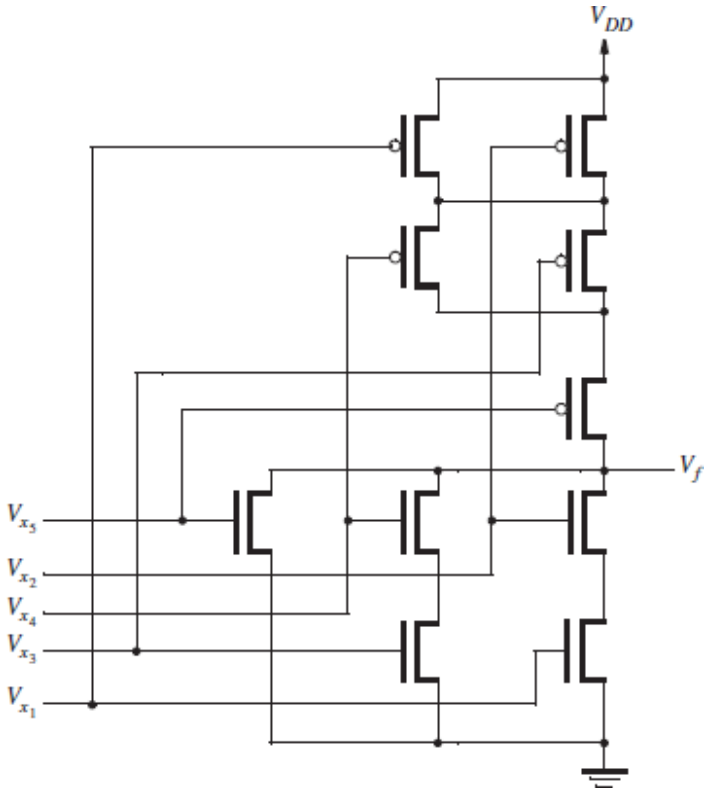


Figure B.76 Circuit for Examples B.13 and B.14.

Solution: Applying Demorgan's theorem in two steps gives

$$\begin{aligned} f &= \overline{x_1 x_2} \cdot \overline{x_3 x_4} \cdot \overline{x_5} \\ &= (\overline{x}_1 + \overline{x}_2) \cdot (\overline{x}_3 + \overline{x}_4) \cdot \overline{x}_5 \end{aligned}$$

Since all input variables are complemented in this expression, we can directly derive the pull-up network as having parallel-connected PMOS transistors controlled by x_1 and x_2, in series with parallel-connected transistors controlled by x_3 and x_4, in series with a transistor controlled by x_5. This circuit, along with the corresponding pull-down network, is shown in Figure B.76.

Example B.14

Problem: For the CMOS complex gate in Figure B.76, determine the sizes of transistors that should be used such that the speed performance of this gate is similar to that of an inverter.

Solution: Recall from Section B.8.5 that a transistor with length L and width W has a drive strength proportional to the ratio W / L. Also recall that when transistors are connected in parallel their widths are effectively added, leading to an increase in drive strength. Similarly, when transistors are connected in series, their lengths are added, leading to a decrease in drive strength. Let us assume that all NMOS and PMOS transistors have the same length, $L_n = L_p = L$. In Figure B.76 the NMOS transistor connected to input V_{x_5} can have the same width as in an inverter, W_n. But the worst-case path through the pull-down network in this circuit involves two NMOS transistors in series. For these NMOS transistors, which are connected to inputs V_{x_1},…, V_{x_4}, we should make the widths equal to $2 \times W_n$. For the pull-up network, the worst-case path involves three transistors in series. Since, as we said in Section B.8.1, PMOS transistors have about half the drive strength of NMOS transistors, we should make the effective width of the PMOS transistors

$$W_p = 3 \times W_n \times 2 = 6W_n$$

Example B.15

Problem: In Section B.8.5, we said that the time needed to charge a capacitor is given by

$$t_p = \frac{C\Delta V}{I}$$

Derive this expression.

Solution: As we stated in Section B.8.5, the voltage across a capacitor cannot change instantaneously. In Figure B.50*b*, as V_f is charged from 0 volts toward V_{DD}, the voltage changes according to the equation

$$V_f = \frac{1}{C}\int_0^{\infty} i(t)\,dt$$

In this expression, the independent variable t is time, and $i(t)$ represents the instantaneous current flow through the capacitor at time t. Differentiating both sides of this expression with respect to time, and rearranging gives

$$i(t) = C\frac{dV_f}{dt}$$

For the case where I is constant, we have

$$\frac{I}{C} = \frac{\Delta V}{\Delta t}$$

Therefore,

$$\Delta t = t_p = \frac{C\Delta V}{I}$$

Example B.16

Problem: In our discussion of Figure B.50*a*, in Section B.8.6, we said that a capacitor, C, that has been charged to the voltage $V_f = V_{DD}$, stores an amount of energy equal to $C\,V^2_{DD}/2$. Derive this expression.

Solution: As shown in Example B.15, the current flow through a charging capacitor, C, is related to the rate of change of voltage across the capacitor, according to

$$i(t) = C\frac{dV_f}{dt}$$

The instantaneous power dissipated in the capacitor is

$$P = i(t) \times V_f$$

Since energy is defined as the power used over a time period, we can calculate the energy, E_C, stored in the capacitor as V_f changes from 0 to V_{DD} by integrating the instantaneous power over time, as follows

$$E_C = \int_0^\infty i(t)V_f\,dt$$

Substituting the above expression for $i(t)$ gives

$$E_C = \int_0^{\infty} C\frac{dV_f}{dt}V_f\,dt$$
$$= C\int_0^{V_{DD}} V_f\,dV_f$$
$$= \frac{1}{2}CV_{DD}^2$$

Example B.17

Problem: In the original NMOS technology, the pull-up device was an *n*-channel MOSFET. But most integrated circuits fabricated today use CMOS technology. Hence it is convenient to implement the pull-up resistor using a PMOS transistor, as shown in Figure B.77. Such a circuit is referred to as a *pseudo-NMOS* circuit. The pull-up device is called a "weak" PMOS transistor because it has a small *W/L* ratio.

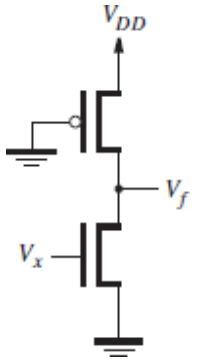


Figure B.77 The pseudo-NMOS inverter.

When $V_x = V_{DD}$, V_f has a low value. The NMOS transistor is operating in the triode region, while the PMOS transistor limits the current flow because it is operating in the saturation region. The current through the NMOS and PMOS transistors has to be equal and is given by Equations B.1 and B.2. Show that the low-output voltage, $V_f = V_{OL}$ is given by

$$V_f = (V_{DD} - V_T)\left[1 - \sqrt{1 - \frac{k_p}{k_n}}\right]$$

where k_p and k_n, called the *gain factors*, depend on the sizes of the PMOS and NMOS transistors, respectively. They are defined by $k_p = k'_p W_p/L_p$ and $k_n = k'_n W_n/L_n$.

Solution: For simplicity we will assume that the magnitude of the threshold voltages for both the NMOS and PMOS transistors are equal, so that

$$V_T = V_{TN} = -V_{TP}$$

The PMOS transistor is operating in the saturation region, so the current flowing through it is given by

$$\begin{aligned} I_D &= \frac{1}{2}k'_p\frac{W_p}{L_p}(-V_{DD} - V_{TP})^2 \\ &= \frac{1}{2}k_p(-V_{DD} - V_{TP})^2 \\ &= \frac{1}{2}k_p(V_{DD} - V_T)^2 \end{aligned}$$

Similarly, the NMOS transistor is operating in the triode region, and its current flow is defined by

$$\begin{aligned} I_D &= k'_n\frac{W_n}{L_n}\left[(V_x - V_{TN})V_f - \frac{1}{2}V_f^2\right] \\ &= k_n\left[(V_x - V_{TN})V_f - \frac{1}{2}V_f^2\right] \\ &= k_n\left[(V_{DD} - V_T)V_f - \frac{1}{2}V_f^2\right] \end{aligned}$$

Since there is only one path for current to flow, we can equate the currents flowing through the NMOS and PMOS transistors and solve for the voltage V_f.

$$\begin{aligned} k_p(V_{DD} - V_T)^2 &= 2k_n\left[(V_{DD} - V_T)V_f - \frac{1}{2}V_f^2\right] \\ k_p(V_{DD} - V_T)^2 - 2k_n(V_{DD} - V_T)V_f + k_nV_f^2 &= 0 \end{aligned}$$

This quadratic equation can be solved using the standard formula, with the parameters

$$a = k_n,\ b = -2k_n(V_{DD} - V_T),\ c = k_p(V_{DD} - V_T)^2$$

which gives

$$V_f = \frac{-b}{2a} \pm \sqrt{\frac{b^2}{4a^2} - \frac{c}{a}}$$
$$= (V_{DD} - V_T) \pm \sqrt{(V_{DD} - V_T)^2 - \frac{k_p}{k_n}(V_{DD} - V_T)^2}$$
$$= (V_{DD} - V_T)\left[1 \pm \sqrt{1 - \frac{k_p}{k_n}}\right]$$

Only one of these two solutions is valid, because we started with the assumption that the NMOS transistor is in the triode region while the PMOS is in the saturation region. Thus

$$V_f = (V_{DD} - V_T)\left[1 - \sqrt{1 - \frac{k_p}{k_n}}\right]$$

Example B.18

Problem: For the circuit in Figure B.77, assume the values $k'_n = 60\ \mu A/V^2$, $k'_p = 0.4k'_n$, $W_n/L_n = 2.0\ \mu m/0.5\ \mu m$, $W_p/L_p = 0.5\ \mu m/0.5\ \mu m$, $V_{DD} = 5$ V, and $V_T = 1$ V. When $V_x = V_{DD}$, calculate the following:

(a) The static current, I_{stat}

(b) The on-resistance of the NMOS transistor

(c) V_{OL}

(d) The static power dissipated in the inverter

(e) The on-resistance of the PMOS transistor

(f) Assume that the inverter is used to drive a capacitive load of 70 fF. Using Equation B.4, calculate the low-to-high and high-to-low propagation delays.

Solution: (*a*) The PMOS transistor is saturated, therefore

$$I_{stat} = \frac{1}{2}k'_p\frac{W_p}{L_p}(V_{DD} - V_T)^2$$
$$= 12\ \frac{\mu A}{V^2} \times 1 \times (5\ V - 1\ V)^2 = 192\ \mu A$$

(*b*) Using Equation B.3,

$$R_{DS} = 1/\left[k'_n \frac{W_n}{L_n}(V_{GS} - V_T)\right]$$
$$= 1/\left[0.060\,\frac{\text{mA}}{\text{V}^2} \times 4 \times (5\,\text{V} - 1\,\text{V})\right] = 1.04\,\text{k}\Omega$$

(*c*) Using the expression derived in Example B.17 we have

$$k_p = k'_p \frac{W_p}{L_p} = 24\,\frac{\mu\text{A}}{\text{V}^2}$$
$$k_n = k'_n \frac{W_n}{L_n} = 240\,\frac{\mu\text{A}}{\text{V}^2}$$

$$V_{OL} = V_f = (5\,\text{V} - 1\,\text{V})\left[1 - \sqrt{1 - \frac{24}{240}}\right]$$
$$= 0.21\,\text{V}$$

(*d*)

$$P_D = I_{stat} \times V_{DD}$$
$$= 192\,\mu\text{A} \times 5\,\text{V} = 960\,\mu\text{W} \approx 1\,\text{mW}$$

(*e*)

$$R_{SDP} = V_{SD}/I_{SD}$$
$$= (V_{DD} - V_f)/I_{stat}$$
$$= (5\,\text{V} - 0.21\,\text{V})/0.192\,\text{mA} = 24.9\,\text{k}\Omega$$

(*f*) The low-to-high propagation delay is

$$t_{p_{LH}} = \frac{1.7C}{k'_p \frac{W_p}{L_p} V_{DD}}$$
$$= \frac{1.7 \times 70\,\text{fF}}{24\,\frac{\mu\text{A}}{\text{V}^2} \times 1 \times 5\,\text{V}} = 0.99\,\text{ns}$$

The high-to-low propagation delay is

$$t_{p_{HL}} = \frac{1.7C}{k'_n \frac{W_n}{L_n} V_{DD}}$$
$$= \frac{1.7 \times 70\,\text{fF}}{60\,\frac{\mu\text{A}}{\text{V}^2} \times 4 \times 5\,\text{V}} = 0.1\,\text{ns}$$

Example B.19

Problem: In Figure B.74 we showed a solution to the static power dissipation problem when NMOS pass transistors are used. Assume that the PMOS pull-up transistor is removed from this circuit. Assume the parameters $k'_n = 60\ \mu A/V^2$, $k'_p = 0.5 \times k'_n$, $W_n/L_n = 2.0\ \mu m/0.5\ \mu m$, $W_p/L_p = 4.0\ \mu m/0.5\ \mu m$, $V_{DD} = 5$ V, and $V_T =$ 1 V. For $V_B = 3.5$ V, calculate the following:

(a) The static current I_{stat}

(b) The voltage V_f at the output of the inverter

(c) The static power dissipation in the inverter

(d) If a chip contains 250,000 inverters used in this manner, find the total static power dissipation.

Solution: (*a*) If we assume that the PMOS transistor is operating in the saturation region, then the current flow through the inverter is defined by

$$I_{stat} = \frac{1}{2}k'_p\frac{W_p}{L_p}(V_{GS} - V_{Tp})^2$$

$$= 120\ \frac{\mu A}{V^2} \times ((3.5\ V - 5\ V) + 1\ V)^2 = 30\ \mu A$$

(*b*) Since the static current, I_{stat}, flowing through the PMOS transistor also flows through the NMOS transistor, then assuming that the NMOS transistor is operating in the triode region, we have

$$I_{stat} = k'_n\frac{W_n}{L_n}\left[(V_{GS} - V_{Tn})V_{DS} - \frac{1}{2}V_{DS}^2\right]$$

$$30\ \mu A = 240\ \frac{\mu A}{V^2} \times \left[2.5\ V \times V_f - \frac{1}{2}V_f^2\right]$$

$$1 = 20V_f - 4V_f^2$$

Solving this quadratic equation yields $V_f = 0.05$ V. Note that the output voltage V_f satisfies the assumption that the PMOS transistor is operating in the saturation region while the NMOS transistor is operating in the triode region. (*c*) The static power dissipated in the inverter is

$$P_S = I_{stat} \times V_{DD} = 30\ \mu A \times 5\ V = 150\ \mu W$$

(*d*) The static power dissipated by 250,000 inverters is

$$250{,}000 \times P_S = 37.5\ \text{W}$$

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

B.1 Consider the circuit shown in Figure PB.1

(a) Show the truth table for the logic function f

(b) If each gate in the circuit is implemented as a CMOS gate, how many transistors are needed?

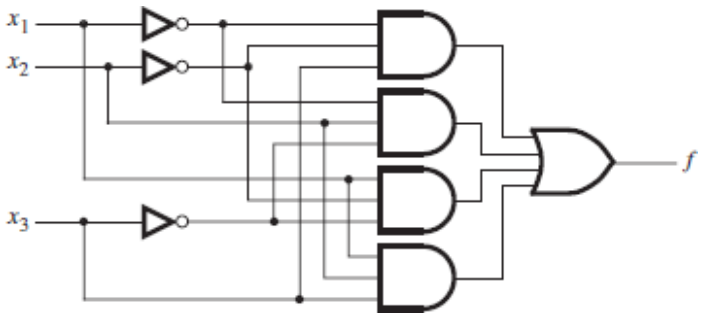


Figure PB.1 A sum-of-products CMOS circuit.

B.2 (a) Show that the circuit in Figure PB.2 is functionally equivalent to the circuit in Figure PB.1

(b) How many transistors are needed to build this CMOS circuit?

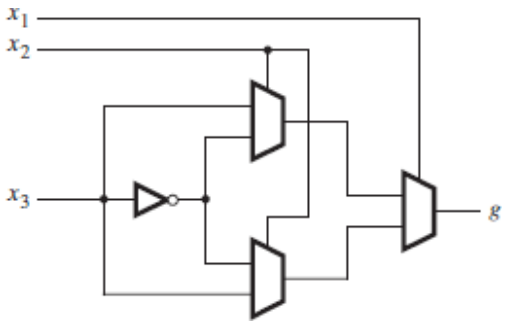


Figure PB.2 A CMOS circuit built with multiplexers.

B.3 (a) Show that the circuit in Figure PB.3 is functionally equivalent to the circuit in Figure PB.2

(b) How many transistors are needed to build this CMOS circuit if each XOR gate is implemented using the circuit in Figure B.55?

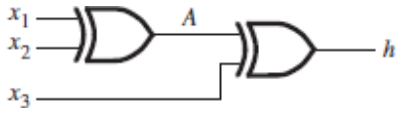


Figure PB.3 Circuit for Problem B.3.

***B.4** In Section B.8.9 we said that a six-input CMOS AND gate can be constructed using two three-input AND gates and a two-input AND gate. This approach requires 22 transistors. Show how you can use only CMOS NAND and NOR gates to build the six-input AND gate, and calculate the number of transistors needed. (*Hint:* Use DeMorgan's theorem.)

B.5 Repeat Problem B.4 for an eight-input CMOS OR gate.

B.6 (a) Give the truth table for the CMOS circuit in Figure PB.4

(b) Derive a canonical sum-of-products expression for the truth table from part (*a*). How many transistors are needed to build a circuit representing the canonical form if only AND, OR, and NOT gates are used?

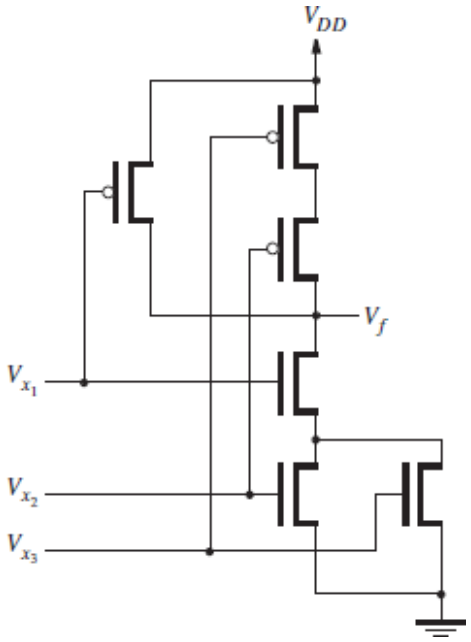


Figure PB.4 A three-input CMOS circuit.

B.7 (a) Give the truth table for the CMOS circuit in Figure PB.5
(b) Derive the simplest sum-of-products expression for the truth table in part (*a*). How many transistors are needed to build the sum-of-products circuit using CMOS AND, OR, and NOT gates?

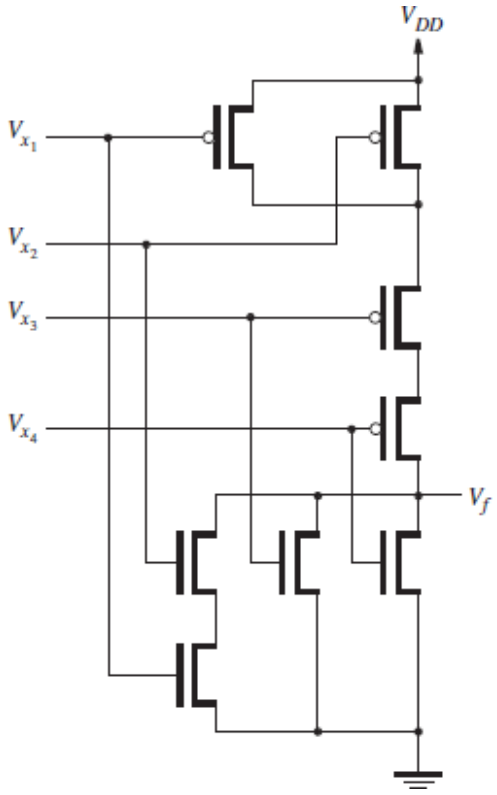


Figure PB.5 A four-input CMOS circuit.

***B.8** Figure PB.6 shows half of a CMOS circuit. Derive the other half that contains the PMOS transistors.

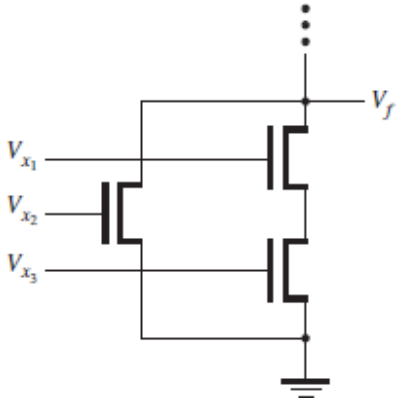


Figure PB.6 The PDN in a CMOS circuit.

B.9 Figure PB.7 shows half of a CMOS circuit. Derive the other half that contains the NMOS transistors.

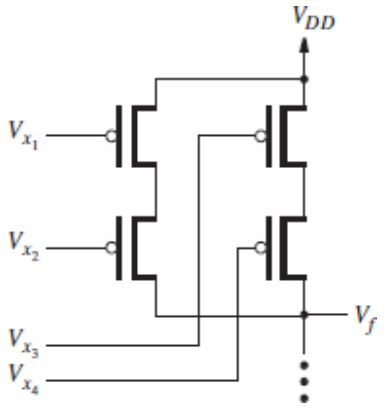


Figure PB.7 The PUN in a CMOS circuit.

B.10 Derive a CMOS complex gate for the logic function $f(x_1, x_2, x_3, x_4) = \sum m(0, 1, 2, 4, 5, 6, 8, 9, 10)$.

B.11 Derive a CMOS complex gate for the logic function $f(x_1, x_2, x_3, x_4) = \sum m(0, 1, 2, 4, 6, 8, 10, 12, 14)$.

***B.12** Derive a CMOS complex gate for the logic function $f = xy + xz$. Use as few transistors as possible (*Hint:* Consider $\bar{f}$).

B.13 Derive a CMOS complex gate for the logic function $f = xy + xz + yz$. Use as few transistors as possible (*Hint:* Consider $\bar{f}$.

***B.14** For an NMOS transistor, assume that $k_n' = 20\ \mu A/V^2, W/L = 2.5\ \mu m/0.5\ \mu m, V_{GS} = 5\ V$,, and $V_T = 1$ V. Calculate

(a) I_D when $V_{DS} = 5$ V

(b) I_D when $V_{DS} = 0.2$ V

B.15 For a PMOS transistor, assume that $k_p' = 10\ \mu A/V^2, W/L = 2.5\ \mu m/0.5\ \mu m, V_{GS} = -5\ V$, and V_T = -1 V. Calculate

(a) I_D when V_{DS} = -5 V

(b) I_D when V_{DS} = -0.2 V

B.16 For an NMOS transistor, assume that $k_n' = 20\ \mu A/V^2, W/L = 5.0\ \mu m/0.5\ \mu m, V_{GS} = 5\ V$, and $V_T = 1$ V. For small V_{DS}, calculate R_{DS}.

***B.17** For an NMOS transistor, assume that $k_n' = 40\ \mu A/V^2, W/L = 3.5\ \mu m/0.35\ \mu m, V_{GS} = 3.3\ V$, and $V_T = 0.66$ V. For small V_{DS}, calculate R_{DS}.

B.18 For a PMOS transistor, assume that $k_n' = 40\ \mu A/V^2$, $W/L = 3.5\ \mu m/0.35\ \mu m$, $V_{GS} = 3.3$ V, and V_T = -1 V. For V_{DS}= -4.8 V, calculate R_{DS}.

B.19 For a PMOS transistor, assume that $k_p' = 16\ \mu A/V^2$, $W/L = 3.5\ \mu m/0.35\ \mu m$, $V_{GS} = -3.3$ V, and V_T = -0.66 V. For V_{DS} = -3.2 V, calculate R_{DS}.

B.20 In Example B.17 we showed how to calculate voltage levels in a pseudo-NMOS inverter. Figure PB.8 depicts a pseudo-PMOS inverter. In this technology, a weak NMOS transistor is used to implement a pull-down resistor.

When V_x = 0, V_f has a high value. The PMOS transistor is operating in the triode region, while the NMOS transistor limits the current flow, because it is operating in the saturation region. The current through the PMOS and NMOS transistors has to be the same and is given by Equations B.1 and B.2. Find an expression for the high-output voltage, $V_f = V_{OH}$, in terms of V_{DD}, V_T, k_p, and k_n, where k_p and k_n are gain factors as defined in Example B.17.

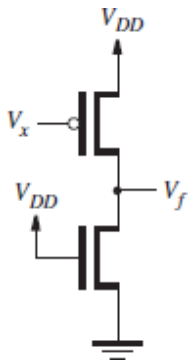


Figure PB.8 The pseudo-PMOS inverter.

B.21 For the circuit in Figure PB.8, assume the values $k_n' = 60\ \mu A/V^2$, $k_p' = 0.4k_n'$, $W_n/L_n = 0.5\ \mu m/0.5\ \mu m$, $W_p/L_p = 4.0\ \mu m/0.5\ \mu m$, $V_{DD} = 5$ V and V_T = 1 V. When V_x = 0, calculate the following:

(a) The static current, I_{stat}

(b) The on-resistance of the PMOS transistor

(c) V_{OH}

(d) The static power dissipated in the inverter

(e) The on-resistance of the NMOS transistor

(f) Assume that the inverter is used to drive a capacitive load of 70 fF. Using Equation B.4, calculate the low-to-high and high-to-low propagation delays.

B.22 Repeat Problem B.21 assuming that the size of the NMOS transistor is changed to $W_n/L_n = 4.0\ \mu\text{m}/0.5\ \mu\text{m}$.

B.23 Example B.17 (see Figure B.77) shows that in the pseudo-NMOS technology the pull-up device is implemented using a PMOS transistor. Repeat this problem for a NAND gate built with pseudo-NMOS technology. Assume that both of the NMOS transistors in the gate have the same parameters, as given in Example B.18.

B.24 Repeat Problem B.23 for a pseudo-NMOS NOR gate.

***B.25** (a) For V_{IH} = 4 V, V_{OH} = 4.5 V, V_{IL} = 1 V, V_{OL} = 0.3 V, and V_{DD} = 5 V, calculate the noise margins NM_H and NM_L

(b) Consider an eight-input NAND gate built using NMOS technology. If the voltage drop across each transistor is 0.1 V, what is V_{OL}? What is the corresponding NM_L using the other parameters from part (*a*).

B.26 Under steady-state conditions, for an *n*-input CMOS NAND gate, what are the voltage levels of V_{OL} and V_{OH}? Explain.

B.27 For a CMOS inverter, assume that the load capacitance is C = 150 fF and V_{DD} = 5 V. The inverter is cycled through the low and high voltage levels at an average rate of f = 75 MHz (a) Calculate the dynamic power dissipated in the inverter (b) For a chip that contains the equivalent of 250,000 inverters, calculate the total dynamic power dissipated if 20 percent of the gates change values at any given time.

***B.28** Repeat Problem B.27 for C = 120 fF, V_{DD} = 3.3 V, and f = 125 MHz.

B.29 In a CMOS inverter, assume that $k_n' = 20\ \mu\text{A/V}^2$, $k_p' = 0.4 \times k_n'$, $W_n/L_n = 5.0\ \mu\text{m}/0.5\ \mu\text{m}$, $W_p/L_p = 5.0\ \mu\text{m}/0.5\ \mu\text{m}$, and V_{DD} = 5 V. The inverter drives a load capacitance of 150 fF (a) Find the high-to-low propagation delay (b) Find the low-to-high low propagation delay (c) What should be the dimensions of the PMOS transistor such that the low-to-high and high-to-low propagation delays are equal? Ignore the effect of the PMOS transistor's size on the load capacitance of the inverter.

B.30 Repeat Problem B.29 for the parameters $k_n' = 40\ \mu\text{A/V}^2$, $k_p' = 0.4 \times k_n'$, $W_n/L_n = W_p/L_p = 3.5\ \mu\text{m}/0.35\ \mu\text{m}$, and V_{DD}=3.3 V.

B.31 In a CMOS inverter, assume that $W_n/L_n = 2$ and $W_p/L_p = 4$. For a CMOS NAND gate, calculate the required W/L ratios of the NMOS and PMOS transistors such that the available current in the gate to drive the output both low and high is equal to that in the inverter.

***B.32** Repeat Problem B.31 for a CMOS NOR gate.

B.33 Repeat Problem B.31 for the CMOS complex gate in Figure B.16. The transistor sizes should be chosen such that in the worst case the available current is at least as large as in the inverter.

B.34 Repeat Problem B.31 for the CMOS complex gate in Figure B.17.

B.35 In Figure B.74 we showed a solution to the static power dissipation problem when NMOS pass transistors are used. Assume that the PMOS pull-up transistor is removed from this circuit. Assume the parameters

$k'_n = 60\ \mu A/V^2, k'_p = 0.4 \times k'_n, W_n/L_n = 1.0\ \mu m/0.25\ \mu m, W_p/L_p = 2.0\ \mu m/0.25\ \mu m, V_{DD} = 2.5\ V$

, and $V_T = 0.6$ V. For $V_B = 1.6$ V, calculate the following:

(a) the static current, I_{stat}

(b) the voltage, V_f, at the output of the inverter

(c) the static power dissipation in the inverter

(d) If a chip contains 500,000 inverters used in this manner, find the total static power dissipation.

B.36 Using the style of drawing in Figure B.70, draw a picture of a PLA programmed to implement $f_1(x_1, x_2, x_3) = \sum$ m(1, 2, 4, 7). The PLA should have the inputs $x_1, \ldots, x_3$; the product terms $P_1, \ldots, P_4$; and the outputs f_1 and f_2.

B.37 Using the style of drawing in Figure B.70, draw a picture of a PLA programmed to implement $f_1(x_1, x_2, x_3) = \sum$ m(0, 3, 5, 6). The PLA should have the inputs $x_1, \ldots, x_3$; the product terms $P_1, \ldots, P_4$; and the outputs f_1 and f_2.

B.38 Show how the function f_1 from Problem B.36 can be realized in a PLA of the type shown in Figure B.71. Draw a picture of such a PLA programmed to implement f_1. The PLA should have the inputs $x_1, \ldots, x_3$; the sum terms $S_1, \ldots, S_4$; and the outputs f_1 and f_2.

B.39 Show how the function f_1 from Problem B.37 can be realized in a PLA of the type shown in Figure B.71. Draw a picture of such a

PLA programmed to implement f_1. The PLA should have the inputs x_1, …, x_3; the sum terms S_1, …, S_4; and the outputs f_1 and f_2.

B.40 Repeat Problem B.38 using the style of PLA drawing shown in Figure B.67.

B.41 Repeat Problem B.39 using the style of PLA drawing shown in Figure B.67.

B.42 Given that f_1 is implemented as described in Problem B.36, list all of the other possible logic functions that can be realized using output f_2 in the PLA.

B.43 Given that f_1 is implemented as described in Problem B.37, list all of the other possible logic functions that can be realized using output f_2 in the PLA.

B.44 Consider the function $f(x_1, x_2, x_3) = x_1\bar{x}_2 + x_1x_3 + x_2\bar{x}_3$. Show a circuit using 5 two-input lookup-tables (LUTs) to implement this expression. As shown in Figure B.39, give the truth table implemented in each LUT. You do not need to show the wires in the FPGA.

***B.45** Consider the function $f(x_1, x_2, x_3) = \sum$ m(2, 3, 4, 6, 7). Show how it can be realized using two two-input LUTs. As shown in Figure B.39, give the truth table implemented in each LUT. You do not need to show the wires in the FPGA.

B.46 Given the function $f = x_1x_2x_4 + x_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_2\bar{x}_3$, a straightforward implementation in an FPGA with three-input LUTs requires four LUTs. Show how it can be done using only 3 three-input LUTs. Label the output of each LUT with an expression representing the logic function that it implements.

B.47 For *f* in Problem B.46, show a circuit of two-input LUTs that realizes the function. You are to use exactly seven two-input LUTs. Label the output of each LUT with an expression representing the logic function that it implements.

B.48 Figure B.39 shows an FPGA programmed to implement a function. The figure shows one pin used for function *f*, and several pins that are unused. Without changing the programming of any switch that is turned *on* in the FPGA in the figure, list 10 other logic functions, in addition to *f*, that can be implemented on the unused pins.

B.49 Assume that a gate array contains the type of logic cell depicted in Figure PB.9. The inputs in_1, …, in_7 can be connected to either 1 or 0, or to any logic signal

(a) Show how the logic cell can be used to realize $f = x_1 x_2 + x_3$ (b) Show how the logic cell can be used to realize $f = x_1 x_3 + x_2 x_3$.

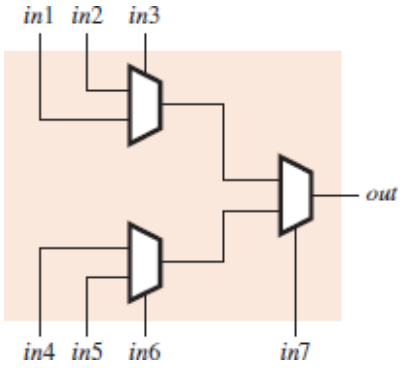


Figure PB.9 A gate-array logic cell.

B.50 Assume that a gate array exists in which the logic cell used is a three-input NAND gate. The inputs to each NAND gate can be connected to either 1 or 0, or to any logic signal. Show how the following logic functions can be realized in the gate array. (*Hint:* Use DeMorgan's theorem.)

(a) $f = x_1 x_2 + x_3$

(b) $f = x_1 x_2 x_4 + x_2 x_3 \overline{x}_4 + \overline{x}_1$

B.51 What logic gate is realized by the circuit in Figure PB.10? Does this circuit suffer from any major drawbacks?

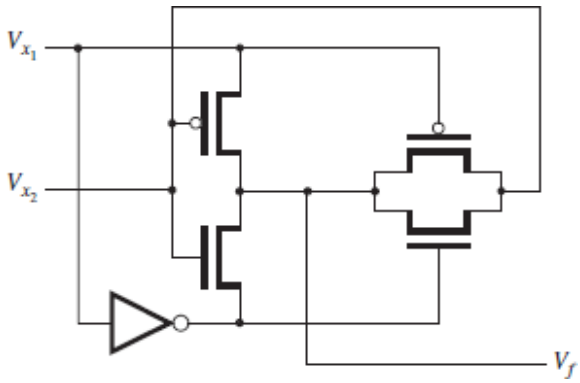


Figure PB.10 Circuit for Problem B.51.

***B.52** What logic gate is realized by the circuit in Figure PB.11? Does this circuit suffer from any major drawbacks?

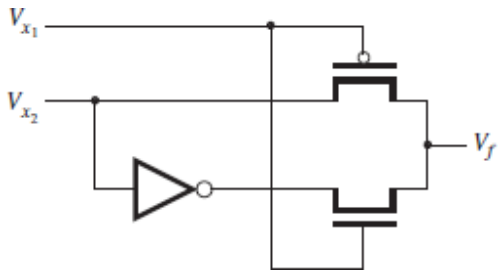


Figure PB.11 Circuit for Problem B.52.

***B.53** Consider the circuit in Figure BP.12. Can this circuit be used as one stage in a ripple-carry adder? Discuss the pros and cons.

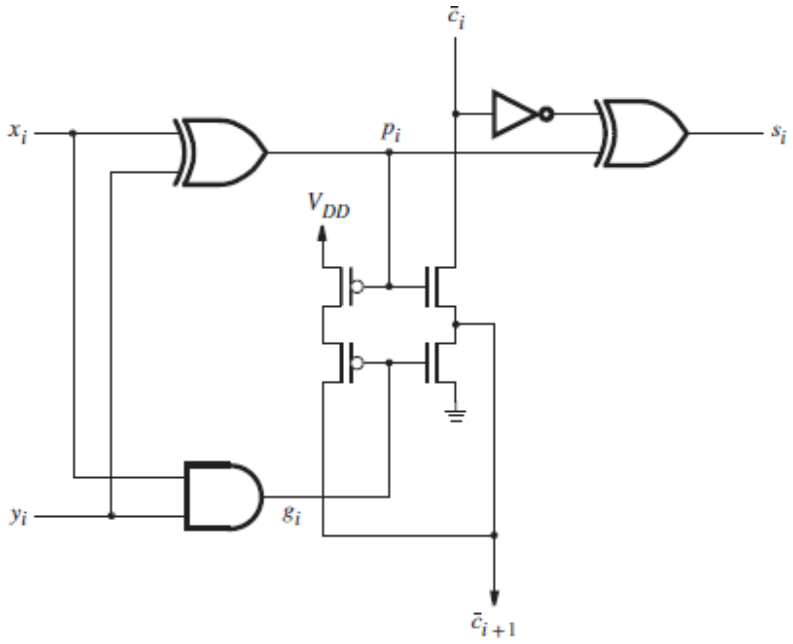


Figure PB.12 Circuit for Problem B.53.

REFERENCES

1. A. S. Sedra and K. C. Smith, *Microelectronic Circuits*, 5th ed. (Oxford University Press: New York, 2003).
2. J. M. Rabaey, *Digital Integrated Circuits*, (Prentice-Hall: Englewood Cliffs, NJ, 1996).

3. Texas Instruments, *Logic Products Selection Guide and Databook CD-ROM*, 1997.
4. National Semiconductor, *VHC/VHCT Advanced CMOS Logic Databook*, 1993.
5. Motorola, *CMOS Logic Databook*, 1996.
6. Toshiba America Electronic Components, *TC74VHC/VHCT Series CMOS Logic Databook*, 1994.
7. Integrated Devices Technology, *High Performance Logic Databook*, 1994.
8. J. F. Wakerly, *Digital Design Principles and Practices* 3rd ed. (Prentice-Hall: Englewood Cliffs, NJ, 1999).
9. M. M. Mano, *Digital Design* 3rd ed. (Prentice-Hall: Upper Saddle River, NJ, 2002).
10. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, CA, 1994).
11. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
12. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).
13. Lattice Semiconductor, Simple PLDs Data Sheets, http://www.latticesemi.com
14. V. C. Hamacher, Z. G. Vranesic, S. G. Zaky, and N. Manjikian, *Computer Organization and Embedded Systems*, 6th ed. (McGraw-Hill: New York, 2011).
15. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design —The Hardware/Software Interface*, 3rd ed. (Morgan Kaufmann: San Francisco, CA, 2004).

answers

CHAPTER 1

1.1. (a) 10100 (b) 1100100 (c) 10000001 (d) 100000100 (e) 10100000000000

1.4. (a) 10001 (b) 100001 (c) 1000011 (d) 10000010 (e) 101000000000 (f) 1100100000000000

1.6. (a) 9 (b) 28 (c) 63 (d) 2730

1.8. (a) 9 (b) 10 (c) 10 (d) 11

CHAPTER 2

2.7. (a) Yes (b) Yes (c) No

2.12. $f = x_1x_3 + x_2x_3 + \bar{x}_2\bar{x}_3$

2.15. $f = (x_1 + x_2)(\bar{x}_2 + x_3)$

2.20. $f = x_2x_3 + x_1\bar{x}_3$

2.23. $f = (x_1 + x_2)(\bar{x}_1 + \bar{x}_3)$

2.28. $f = x_1x_2 + x_1x_3 + x_2x_3$

2.32. $f = (x_1 + x_2 + \bar{x}_3)(x_1 + \bar{x}_2 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + x_2 + x_3)$

2.33. $f = \bar{x}_1x_3 + \bar{x}_1x_2 + x_2x_3 + x_1\bar{x}_2\bar{x}_3$

2.37. SOP form: $f = \bar{x}_1x_2 + \bar{x}_2x_3$

POS form: $f = (\bar{x}_1 + \bar{x}_2)(x_2 + x_3)$

2.38. SOP form: $f = x_1\bar{x}_2 + x_1x_3 + \bar{x}_2x_3$

POS form: $f = (x_1 + x_3)(x_1 + \bar{x}_2)(\bar{x}_2 + x_3)$

2.41. SOP form: $f = \bar{x}_3\bar{x}_5 + \bar{x}_3x_4 + x_2x_4\bar{x}_5 + \bar{x}_1x_3\bar{x}_4x_5 + x_1x_2\bar{x}_4x_5$

POS form: $f = (\bar{x}_3 + x_4 + x_5)(\bar{x}_3 + \bar{x}_4 + \bar{x}_5)(x_2 + \bar{x}_3 + \bar{x}_4)(x_1 + x_3 + x_4 + \bar{x}_5)(\bar{x}_1 + x_2 + x_4 + \bar{x}_5)$

2.45. $f = x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4$

2.47. The statement is false. As a counter example consider $f(x_1, x_2, x_3) = \sum m(0, 5, 7)$. Then, the minimum-cost SOP form $f = x_1x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$ is unique But, there are two minimum-cost POS forms:

$$f = (x_1 + \bar{x}_3)(\bar{x}_1 + x_3)(x_1 + \bar{x}_2) \quad \text{and}$$
$$f = (x_1 + \bar{x}_3)(\bar{x}_1 + x_3)(\bar{x}_2 + x_3)$$

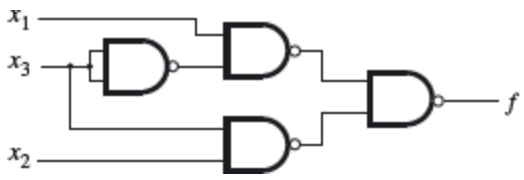
2.48. In a combined circuit:

$$f = \bar{x}_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_3\bar{x}_4 + x_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1x_2x_3$$
$$g = \bar{x}_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_3\bar{x}_4 + x_1\bar{x}_2\bar{x}_3x_4 + x_1x_2x_4$$

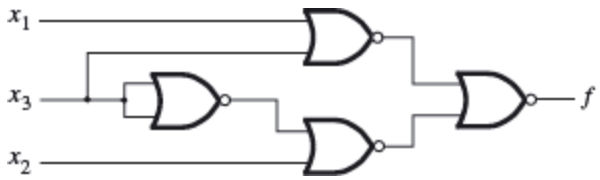
The first 3 product terms are shared; hence the total cost is 31.

2.50. $f = (x_1 + x_4 + x_5) \cdot (x_1 + x_2 + x_3) \cdot (\bar{x}_1 + \bar{x}_2 + x_3) \cdot (\bar{x}_1 + \bar{x}_4 + x_5)$

2.54. The circuit is



2.56. The circuit is



2.66. Representing both functions in the form of Karnaugh map, it is easy to show that $f = g$.

CHAPTER 3

3.1. (a) 478 (b) 743 (c) 2025 (d) 41567 (e) 61680

3.2. (a) 478 (b) –280 (c) –1

3.3. (a) 478 (b) –281 (c) –2

3.4. The numbers are represented as follows:

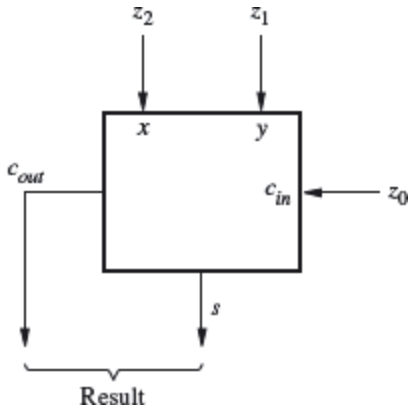
Decimal	Sign and Magnitude	1's Complement	2's Complement
73	000001001001	000001001001	000001001001
1906	011101110010	011101110010	011101110010
–95	100001011111	111110100000	111110100001
–1630	111001011110	100110100001	100110100010

3.12. From Equation 3.4, each c_i requires i AND gates and one OR gate. Therefore, to determine all c_i signals we need $\sum_{i=1}^{n}(i+1) = (n^2+3n)/2$ gates. In addition to this, we need $3n$ gates to generate all g, p, and s functions. Therefore, a total of $(n^2+9n)/2$ gates are needed.

3.13. 84 gates

3.17. The code in Figure P3.2 represents a multiplier. It multiplies the lower two bits of *Input* by the upper two bits of *Input*, producing the four-bit *Output*.

3.21. A full-adder circuit can be used, such that two of the bits of the number are connected as inputs x and y, while the third bit is connected as the carry-in. Then, the carry-out and sum bits will indicate how many input bits are equal to 1. The circuit is

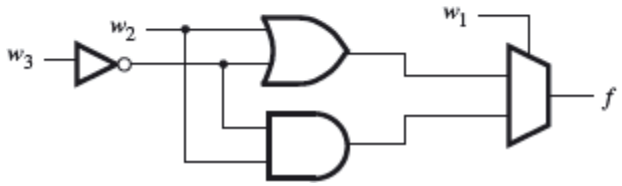


CHAPTER 4

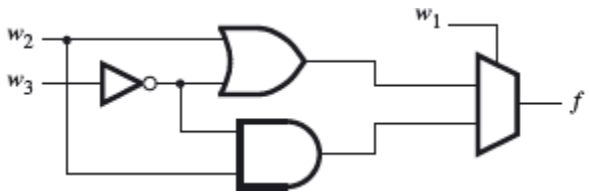
4.3.

w_1	w_2	w_3	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

w_1	f
0	$w_2 + \overline{w}_3$
1	$w_2\overline{w}_3$



4.5. The derived circuit is



4.10. $f(w_1, w_2, \ldots, w_n) = [w_1 + f(0, w_2, \ldots, w_n)] \cdot [\overline{w}_1 + f(1, w_2, \ldots, w_n)]$

4.11. Expansion of f in terms of w_2 gives

$$\begin{aligned} f &= \overline{w}_2(\overline{w}_1 + \overline{w}_3) + w_2(w_1 w_3) \\ &= w_2 \oplus (\overline{w}_1 + \overline{w}_3) \\ &= w_2 \oplus \overline{w_1 w_3} \end{aligned}$$

The cost of this circuit is 2 gates + 4 inputs = 6.

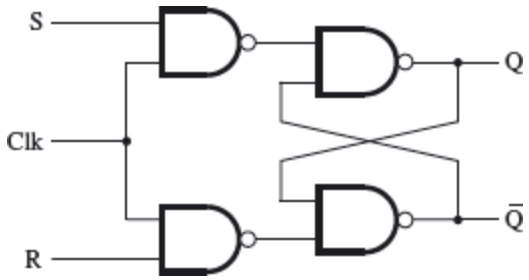
4.13.

$$a = w_3 + w_2 w_0 + w_1 + \overline{w}_2 \overline{w}_0$$
$$b = \overline{w}_1 \overline{w}_0 + w_1 w_0 + \overline{w}_2$$
$$c = w_2 + \overline{w}_1 + w_0$$

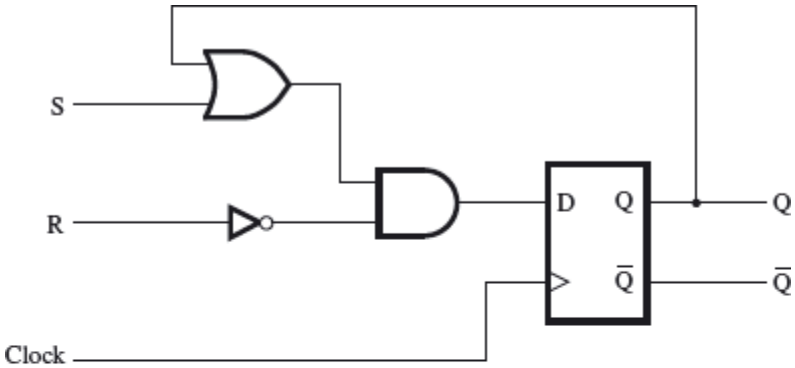
4.18. The code in Figure P4.2 is a 2-to-4 decoder with an enable input. It is not a good style for defining this decoder. The code is not easy to read. Moreover, the VHDL compiler often turns IF statements into multiplexers, in which case the resulting decoder may have multiplexers controlled by the *En* signal on the output side.

CHAPTER 5

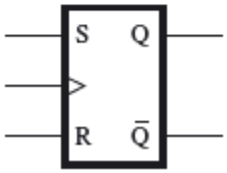
5.3. The circuit is



5.5. A possible circuit is

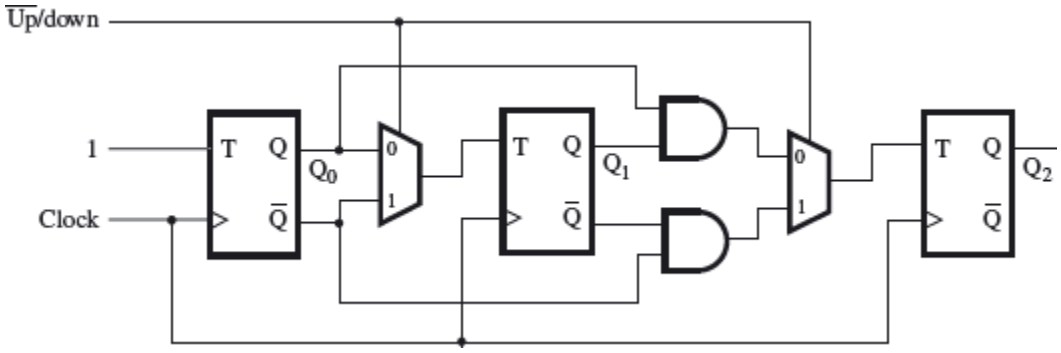


S	R	Q(t + 1)
0	0	Q(t)
0	1	0
1	0	1
1	1	0



5.8. The circuit acts as a negative-edge-triggered JK flip-flop, in which $J = A$, $K = B$, Clock = C, Q = D, and $\overline{Q} = E$.

5.15. The following circuit implements the desired counter



5.17. The counting sequence is 000, 001, 010, 111.

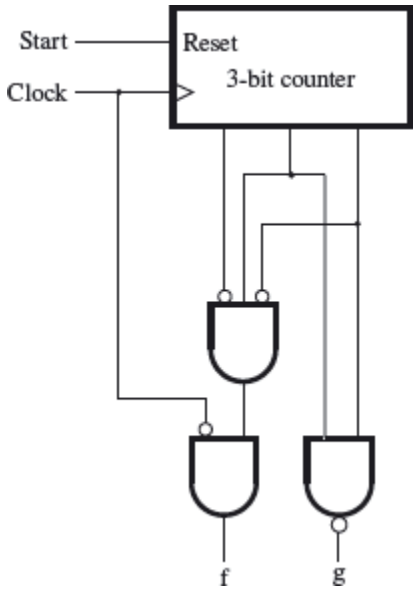
5.23. The longest delay in the circuit is the from the output of FF_0 to the input of FF_3. This delay totals 5 ns. Thus the minimum period for which the circuit will operate reliably is

$$T_{min} = 5 + 3 + 1 = 9 \text{ ns}$$

The maximum frequency is

$$F_{max} = 1/T_{min} = 111 \text{ MHz}$$

5.29. A suitable circuit is



CHAPTER 6

6.1. The expressions for the inputs of the flip-flops are

$$D_2 = Y_2 = \overline{w}y_2 + \overline{y}_1\overline{y}_2$$
$$D_1 = Y_1 = w \oplus y_1 \oplus y_2$$

The output equation is $z = y_1\, y_2$.

6.2. The expressions for the inputs of the flip-flops are

$$J_2 = \overline{y}_1$$
$$K_2 = w$$
$$J_1 = \overline{w}y_2 + w\overline{y}_2$$
$$K_1 = J_1$$

The output equation is $z = y_1\, y_2$.

6.5. Minimal state table is

Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	A	B	0
B	E	C	0
C	D	C	0
D	A	F	1
E	A	F	0
F	E	C	1

6.6. Minimal state table is

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	D	C	0	0
C	D	C	1	0
D	A	B	0	1

6.12. Minimal state table is

Present state	Next state		Output p
	$w = 0$	$w = 1$	
A	B	C	0
B	D	E	0
C	E	D	0
D	A	F	0
E	F	A	0
F	B	C	1

6.15. The next-state expressions are

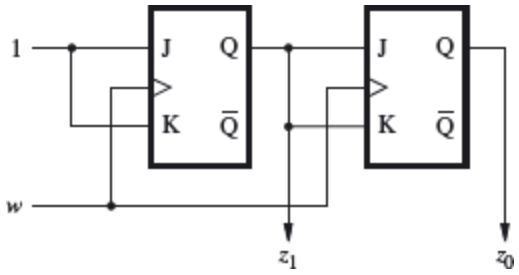
$$
\begin{aligned}
D_4 = Y_4 &= \overline{w}y_3 + wy_1 \\
D_3 = Y_3 &= \overline{w}(y_1 + y_4) \\
D_2 = Y_2 &= \overline{w}y_2 + wy_4 \\
D_1 = Y_1 &= w(y_2 + y_1)
\end{aligned}
$$

The output is given by $z = y_4$.

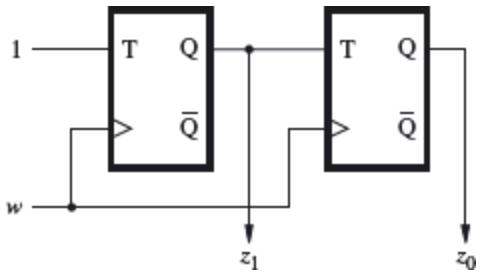
6.17. Minimal state table is

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	C	0	0
C	F	C	0	1
F	C	A	0	1

6.21. The desired circuit is



6.22. The desired circuit is



6.29. The state table is

Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	A	C	0
B	A	D	1
C	A	D	0
D	A	B	0

The circuit produces $z = 1$ whenever the input sequence on w comprises a 0 followed by an even number of 1s.

CHAPTER 8

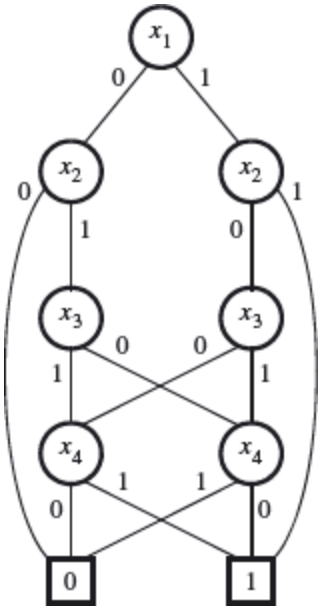
8.1. $f = (x_3 \uparrow g) \uparrow ((g \uparrow g) \uparrow x_4)$ where $g = (x_1 \uparrow (x_2 \uparrow x_2)) \uparrow ((x_1 \uparrow x_1) \uparrow x_2)$

8.2. $\bar{f} = ((x_3 \downarrow x_3) \downarrow g) \downarrow ((g \downarrow g) \downarrow (x_4 \downarrow x_4))$, where $g = ((x_1 \downarrow x_1) \downarrow x_2) \downarrow (x_1 \downarrow (x_2 \downarrow x_2))$. Then, $f = \bar{f} \downarrow \bar{f}$.

8.5. $f = \bar{x}_1(x_2 + x_3)(x_4 + x_5) + x_1(\bar{x}_2 + x_3)(\bar{x}_4 + x_5)$

8.8. $f = g \cdot h + \bar{g} \cdot \bar{h}$, where $g = x_1x_2$ and $h = x_3 + x_4$

8.12. The BDD is



8.18. $f = \bar{x}_1\bar{x}_2\bar{x}_4 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_2x_3x_4$

CHAPTER 9

9.1. The flow table is

Present state	Next state				z_2z_1
	$w_2w_1 = 00$	01	10	11	
A	D	C	D	C	11
B	D	D	(B)	(B)	10
C	D	(C)	D	(C)	01
D	(D)	C	B	C	00

The behavior is the same as described in the flow table in Figure 9.21*a*, if the state interchanges A ↔ D and B ↔ C are made.

9.8. Using the merger diagram in Figure 9.40*a*, the FSM in Figure 9.39 becomes

Present state	Next state				Output z
	$w_2w_1 = 00$	01	10	11	
A	(A)	G	E	–	0
B	(B)	C	(B)	D	0
C	B	(C)	E	(C)	1
D	–	C	E	(D)	0
E	A	–	(E)	D	1
G	B	(G)	–	D	1

9.10. The minimum-cost hazard-free implementation is

$$f = \bar{x}_1\bar{x}_3\bar{x}_4 + x_1x_2x_4 + x_1x_3x_4$$

9.12. The minimum-cost hazard-free POS implementation is

$$f = (x_1 + x_2 + x_4)(x_1 + x_2 + \bar{x}_3)(x_1 + \bar{x}_3 + \bar{x}_4)(x_2 + \bar{x}_3 + x_4)$$

9.14. If $A = B = D = E = 1$ and C changes from 0 to 1, then f changes 0 → 1 → 0 and g changes 0 → 1 → 0 → 1. Therefore, there is a static hazard on f and a dynamic hazard on g.

9.17. The excitation table is

Present state y	Next state				Output			
	$wc = 00$	01	10	11	00	01	10	11
	Y				z			
0	(0)	(0)	1	(0)	0	0	0	0
1	0	(1)	(1)	(1)	0	1	0	1

The next-state expression is $Y = w\bar{c} + cy + wy$. Note that the term wy is included to prevent a static hazard. The output expression is $z = cy$.

CHAPTER 11

11.1. A minimal test set must include the tests $w_1\ w_2w_3$ = 011, 101, and 111, as well as one of 000, 010, or 100.

11.3. The two functions differ only in the vertex $x_1x_2x_3x_4$ = 0111. Therefore, the circuits can be distinguished by applying this input valuation.

11.5. The tests are $w_1w_2w_3w_4$ = 1111, 1110, 0111, and 1111.

11.9. Cannot detect if the input wire w_1 is stuck-at-1. The reason is that this circuit is highly redundant. It realizes the function $f = w_3(\overline{w}_1 + \overline{w}_2)$, which can be implemented with a simpler circuit.

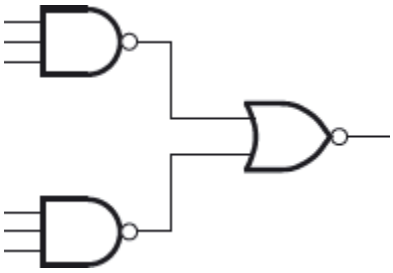
11.11. Test set = {0000, 0111, 1111, 1000}. It would work with XORs implemented as shown in Figure 4.26*c*. For *n* bits, the same patterns can be used; thus

$$\text{Test set} = \{00\ldots00, 011\ldots1, 11\ldots1, 100\ldots0\}.$$

11.12. In the decoder circuit in Figure 6.16*c* the four AND gates are enabled only if the *En* signal is active. The required test set has to include all four valuations of w_1 and w_2 when *En* = 1. It is also necessary to test if the *En* wire is stuck at 1, which can be accomplished with the test $w_1w_2En = 000, 001, 011, 101$, w_1w_2En = 000. Therefore, a complete test set comprises w_1w_2En = 000, 001, 011, 101, and 111.

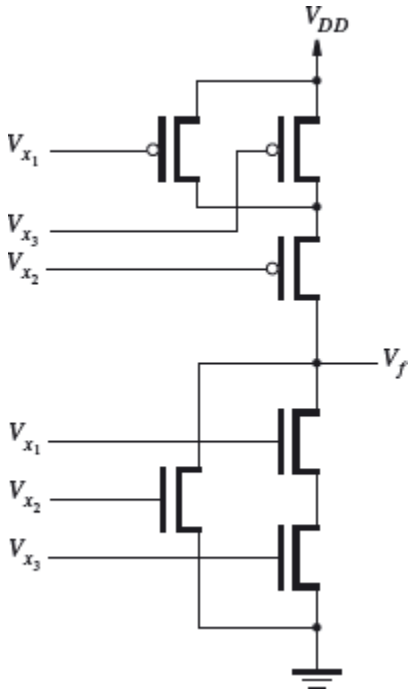
APPENDIX B

B.4. Using the circuit

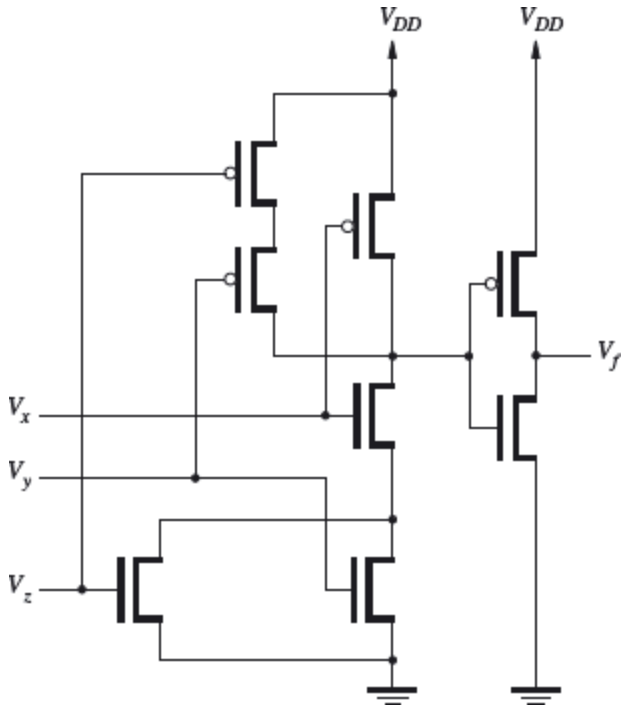


The number of transistors needed is 16.

B.8. The complete circuit is



B.12. The required circuit is



B.14. (a) $I_D = 800\ \mu A$ (b) $I_D = 78\ \mu A$

B.17. $R_{DS} = 947\ \Omega$

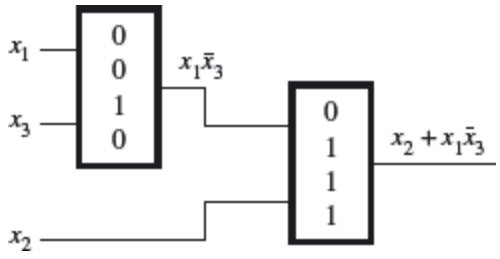
B.25. (a) $NM_H = 0.5\ V$ $NM_L = 0.7\ V$ (b) $V_{OL} = 0.8\ V$ $NM_L = 0.2\ V$

B.28. (a) $P_{NOT\_gate} = 163\ \mu W$ (b) $P_{total} = 8.2\ W$

B.32. The two NMOS transistors in a CMOS NOR gate are connected in parallel. The worst case current to drive the output low happens when only one of these transistors is turned "ON". Thus each transistor has to have the same dimensions as the NMOS transistor in the inverter, namely $W_n/L_n = 2$.

The two PMOS transistors are connected in series. If each of these transistors had the ratio W_p/L_p, then the two transistors could be thought of as one transistor with a $W_p/2\ L_p$ ratio. Thus each PMOS transistor must be made twice as wide as that in the inverter, namely $W_n/L_n = 8$.

B.45 $f = x_2 + x_1\bar{x}_3$. The corresponding circuit is



B.52 The circuit in Figure PB.11 is a two-input XOR gate. This circuit has two drawbacks: when both inputs are 0 the PMOS transistor must drive f to 0, resulting in $f = V_T$ volts. Also, when $x_1 = 1$ and $x_2 = 0$, the NMOS transistor must drive the output high, resulting in $f = V_{DD} - V_T$.

B.53 Yes, it works. The NOT gate that produces c_i is not needed in stages where $i > 0$. The drawback is "poor" propagation of $\bar{c}_i = 1$ through the topmost NMOS transistor. The positive aspect is fewer transistors needed to produce $\bar{c}_{i+1}$.

INDEX

A

Absorption property, 32
Accumulator, 690
Active clock edge, 310
Active-low signal, 760
Adder:
  BCD, 162–163
  carry-lookahead, 138–143
  full-adder, 121, 122
  half-adder, 119
  ripple-carry, 122–123
  serial, 343
  in VHDL code, 144–147
Adder/subtractor unit, 131–132
Addition:
  BCD, 160–163
  carry, 119
  generate function, 139
  1's complement, 128–129
  overflow, 136–137
  propagate function, 139
  sign-and-magnitude, 128
  sum, 119
  2's complement, 129–130
  of unsigned numbers, 118–124
  VHDL, 149
Algorithm, 420
  for division, 433
  for multiplication, 426
Algorithmic state machine (ASM):
  ASM block, 383
  ASM charts, 380–383, 427, 434, 443
  conditional output box, 381
  decision box, 381
  implied timing, 420–421
  state box, 380
Aliasing problem, 637
Analysis, 376
  multilevel circuits, 478–483
AND gate. *See* Logic gates
Anode, 286

Application-specific integrated circuits (ASICs), 5, 736
Arbiter circuit, 371–375, 540–546
Architecture (VHDL), 66, 658–660
  body, 658
  declarative part, 658
Arithmetic:
  floating-point. *See* Floating-point numbers
  mean, 442–445
  operators (VHDL), 212
  overflow, 136–137
  See also Addition; Division; Multiplication; Subtraction
Arithmetic and logic unit (ALU), 209
Arithmetic assignment (VHDL), 149–152
Arithmetic circuits, design of:
  schematic capture, 144
  VHDL, 144–147
Array (VHDL), 655–656
Array multiplier, for unsigned numbers, 154
ASCII code, 14, 15
ASICs (application-specific integrated circuits), 5, 736
ASM block, 383
ASM chart, 380–383, 427, 435, 443
Aspect ratio, 761
Associative property, 32
Asynchronous clear (in VHDL), 271, 682
Asynchronous clear (reset), 245
Asynchronous counters, 252–254
Asynchronous reset, 260
Asynchronous sequential circuits, 522–597
  analysis of, 525–533
  modulo-4 counter, 538–540
  synthesis of, 533–546
Attribute (VHDL), 270, 681
  enum_encoding, 339
  EVENT, 270
Axioms of Boolean algebra, 31

B

Ball grid array (BGA), 733
Barrel shifter, 221, 432
Basic latch, 232–234
BCD. *See* Binary-coded decimal (BCD)
BCD-to-7-segment decoder, 191–192, 211
Behavioral VHDL code, 193–194, 291
BILBO (built-in logic block observer), 637–639
Binary-coded decimal (BCD), 159–163
  additon, 160–163
  counters, 261–262
  digits, 159

Binary decision diagrams (BDDs), 466, 487–493
  derivation, using Shannon's expansion, 488–493
  packages, 493
  practical use of, 493
Binary decoder. *See* Decoder
Binary encoder. *See* Encoder
Binary numbers, 11–13
  conversion to decimal number, 13–14
  in VHDL code, 650
BIST (built-in self-test), 633–637
Bit, 12
BIT (data type), 65, 67, 651
Bit-counting circuit, 420–425
BIT_VECTOR type, 651
Body effect, 752
Boole, George, 30
Boolean algebra, 30–31
  axioms of, 31
  notation and terminology, 38–39
  properties of, 32–34, 494
BOOLEAN type, 654
Boundary scan, 640
Branching, 499, 506
Buffer, 719, 757–759
  inverting, 757
  tri-state, 400–402, 759–761
  VHDL (port mode), 280
Built-in logic block observer (BILBO), 637–639
Built-in self-test (BIST), 633–637
Bus, 400
  multiplexers to implement, 402–404
  structures, VHDL code for, 404–408
Bypass capacitors, 641

C

CAD. *See* Computer-aided design (CAD)
Canonical expressions:
  canonical product-of-sums, 48–50
  canonical sum-of-products, 45–47
Canonical sum-of-products, 45–47
Capacitance, 746
Carry, 119
  carry-in, 120
  carry-out, 120
Carry-lookahead adder, 138–143
CASE statement, 208, 416–419, 672–673
Cathode, 286
Channel (in MOSFET), 741
Characteristic impedance, 642

Characteristic table, 233
Chip configuration, 63
Chip fabrication, 63
Chip layout, 736
Circuit ground, 21
Clear input, 243–245, 255–256
Clock distribution, 456–457
Clock divider, 284
Clock signal, 236
Clock skew, 294–296
Clock synchronization, 253, 454–461
Clock-to-output time (t_{CO}), 268
Clock-to-Q propagation delay (t_{CQ}), 246, 457
CMOS logic circuits, 619
Code:
  BCD. *See* Binary-coded decimal
  converter, 191–192
  Gray, 217
Cofactor, 182
Coincidence operation, 122
Column dominance, 497
Combinational circuits, 173–229, 619, 629
Combining property, 32, 73–74, 494
Comment (VHDL), 650
Commutative property, 32
Comparator, 192
Compatible states, 549
Complement:
  diminished-radix, 135
  of logic variable, 23
  9's, 133
  1's, 126
  radix-complement schemes, 132–136
  10's, 133
  2's, 126–128
Complementary MOS (CMOS) technology, 704, 707–715
  inverter, 744–745
Completely specified FSM, 360
Complex gate (CMOS), 711
Complex programmable logic device (CPLD), 727–731
Component (VHDL), 661
COMPONENT declaration statement (VHDL), 146, 661–663
Computer, structure of, 7–10
Computer-aided design (CAD), 7
  complete design flow, 63
  design entry, 60–61
  functional simulation, 62
  physical design, 62
  synthesis, 604–609
  technology mapping, 606–609

timing analysis, 62–63, 292–296
tools. *See* Tools, CAD
typical system, 605
Concatenate operator, 151
Concatenation (VHDL), 212–213
Concurrent assignment statement (VHDL), 202–203, 664–668
Conditional signal assignment (VHDL), 161, 197–198, 667–668
Consensus, 37
Consensus property, 32
Consistency check, 623
Constant (VHDL), 650, 654–655
Control circuit, 400, 422–423, 428–429, 435
Conversion of types (VHDL), 655
Cost, of logic circuit, 83
Counters, 252
asynchronous, 252–254
asynchronous sequential circuits, 538–540
binary-coded-decimal, 261–262
design, 265
down-counter, 253–254, 280
Johnson, 265
Counters (*cont...*)
modulo-n, 362
with parallel load, 258–260
reset, 260–261
ring, 262–264
ripple, 253
synchronous, 254–258, 362
up-counter, 252–253, 278–279, 414
VHDL code, 279–280, 688
VHDL sequential statements for, 273–283
Cover, of function, 82–83
Critical path, 294, 614
Critical-path delay, 138
Crossbar, 175
Crosstalk, 641
Cubical notation, 483–486
optimization techniques based on, 493–509
Current flow:
dynamic, 749
leakage, 744, 749–750
short-circuit, 750
static, 749
Custom chips, 736–738
Custom-designed chips, 5
Cut off region, 740
Cut set, 533

D

D-algorithm, 625
Datapath circuit, 400, 421–422, 427–428, 434–435, 439, 444–445
Data types (VHDL), 67–69
DC-set, 506
Decimal numbers, 11
  binary number conversion to, 13–14
Decision tree, 487
Decoder, 184–188, 411
  tree, 186
Decomposed full-adder, 122
Default value (VHDL), 416, 423, 674
Delay. *See* Propagation delays
DeMorgan's theorem, 32–33, 51
Demultiplexer, 188
Design entry, 60–61
Design for testability, 629–633
Design process, 5–7
D flip-flops, 239–247, 681
  alternative code for, 270–271
  with clear and preset, 243–245
  code for, 270
  edge-triggered, 239–240
  synchronous counters with, 257–258
Digital hardware, 2–4
Digital representation of information, 10–16
Digital system, 400
Digital-to-analog (D/A) converter, 14
Diminished-radix complement, 135
DIP (dual-inline package), 718
Disjoint decomposition, 473
Distributive property, 32, 37–38
Divider, 432–442
  ASM chart for, 434
  circuit, enhancements to, 435–439
  VHDL, 439–442
Division, 433
Don't-care conditions, 88–90
Double precision. *See* Floating-point numbers
Double-precision floating-point numbers, 158–159
Down-counter, 253–254, 280
Drain (in MOSFET transistor), 702
Drive capability, 757
Dual-inline package (DIP), 718
Duality, 31–32
Dynamic hazards, 581–583
Dynamic power, 749

E

Edge triggered, 239, 241

Electrically-erasable programmable read-only memory (EEPROM), 765–767
Electrical semiconductor, 739
1149.1 [6] Standard, IEEE, 640
Enable inputs, 255–256, 347
  flip-flops and registers with, 280–283
  shift registers with, 283
Encoder:
  binary, 189–190
  priority, 190
Entity, 66, 657–660
enum_encoding, 339
ENUMERATION type, 654
Equivalence:
  of logic networks, 28–29
  of states, 351, 353
Equivalent gates, 731
Erasable PROM (EPROM), 767
Errors in VHDL code, 696–699
Essential prime implicant, 84
EVENT attribute, 270
Excitation table, 366, 525
Exclusive-NOR (XNOR) gate. *See* Logic gates
Exclusive-OR (XOR) gate. *See* Logic gates
Expansion theorem (Shannon's), 181–182
Exponent, 157
Extern signal, 402

F

Factoring, 467–469
Fall time, 748
Fan-in, 143, 467–469, 755–757
Fan-out, 757
Fast adders, 138–143
Fault:
  detection of specific, 623–625
  model, 618–619
  multiple, 619
  propagation, 623–625
  single, 619
  stuck-at model, 618
Feedback, 232, 248
Feedback loop, 755
Field-programmable gate array (FPGA), 5, 10, 731–736
Field-programmable logic array (FPLA), 765
Finite state machine (FSM), 310
  incompletely specified, 360
  summary of design procedure, 318–319
Fixed-point numbers, 157
Flip-flops:

asynchronous inputs to, 458–459
edge-triggered D. *See* D flip-flops
with enable inputs, 280–283
JK, 248–249, 365–368
M-S D, 240–241, 527–532
T, 247–248
timing parameters, 246–247, 457–458
VHDL code for, 681
Floating gate, 765
Floating-point numbers, 157–159
double-precision, 158–159
exponent, 157
IEEE standard, 157–158
mantissa, 157
normalized, 157
single-precision, 157–158
Flow table, 525
primitive, 548
state reduction, 546–559
F_{max}, *(maximum, clock, frequency)*, 292
FOR GENERATE statement, 200–201, 278, 668
FOR LOOP statement, 278, 673
Fowler-Nordheim tunneling, 767
FPGA (field-programmable gate array), 5, 10
FSM. *See* Finite state machine
Full-adder, 121, 122
Functional behavior, 28
Functional decomposition, 470–475
Functional equivalence, 28–29
Functional simulation, 62
Fundamental mode, 522

G

Gate (in MOSFET transistor), 702
Gate arrays, 736–738
Gate capacitance, 747
Gated D latch, 237–239, 526–528, 680–681
Gate delay. *See* Propagation delays
Gated SR latch, 234–237
Gate optimization, 604
Gates. *See* Logic gates
GENERATE statement, 200–201, 668
GENERIC, 274, 668–669
GENERIC MAP, 691
Glitch, 576
Global signals, 334–335
Gray code, 217

H

Half-adder, 119
Hamming distance, 560
Handshake signaling, 540
Hardware description language (HDL), 60–61
Hazards, 527, 576–583
  dynamic, 581–583
  significance of, 583
  static, 576–581
Heuristic approach, 84
Hexadecimal numbers, 116–118
Hierarchical design, 60
Hierarchical VHDL code, 70–73, 196
High impedance state, 400
Hold time (t_h), 239
H tree, 456
Huntington's basic postulates, 33
Hypercube, 486

I

IEEE (Institute of Electrical and Electronics Engineers), 60–61
IEEE Standards, 60–61
  1076, 63
  1149.1 [6], 640
  1164, 63
  floating-point numbers, 157
  Verilog, 61
IF GENERATE statement, 201, 668
IF statement, 202, 671–672
IF-THEN-ELSE statement, 408
Implicant, of function, 82
Implied memory (VHDL), 207, 416, 674–675
Incompletely specified FSM, 360
Incompletely specified functions, 88–90
Information, digital representation of, 10–16
Input variable, 21
Instantiation (of VHDL components), 146, 661–663
Instrumentation, 643–644
In-system programming (ISP), 727
Integer:
  signed, 124–138
  unsigned, 116, 118–124
  in VHDL, 152, 279–280, 653–654
INTEGER type (VHDL), 152, 279, 653–654
Integrated circuit technology, 701–773
International Roadmap for Devices and Systems (IRDS), 3
Intersection, 35–36
Inversion, 23–24

Inverter, 706
  CMOS, 744–745

J

JK flip-flop, 248–249, 365–368
Johnson counters, 265
JTAG port, 729–730

K

Karnaugh maps, 73–81
  five-variable, 80–81
  four-variable, 78–80
  three-variable, 77–78
  two-variable, 76–77
k-cube, 486
k-successor, 353

L

Large-scale integration (LSI), 720
Last-significant bit (LSB), 12
Latch:
  basic, 232–234
  basic SR, 523
  gated D, 237–239, 526–528, 680–681
  gated SR, 234–237
  VHDL code, 268–273
Leakage current, 744, 749–750
LED (light-emitting diode), 284–288
Level sensitive element, 239, 241
Level-sensitive scan design, 633
LFSRs (linear feedback shift registers), 634–635, 637
Libraries:
  VHDL, 146–147
  work, 147, 661
Light-emitting diode (LED), 284–288
Linear feedback shift registers (LFSRs), 634–635, 637
Linear region, 741
Literal, 82
Logical operators (VHDL), 211–212
Logical product (AND), 39
Logical sum (OR), 39
Logic analyzer, 643–644
Logic circuits, 20
  behavioral specification of, 69–70
  CMOS, 619
  design examples, 54–59

structural specification of, 65–69
testing, 618–644
Logic expression, 21
Logic function, 21
AND, 22
alternative representations of, 483–493
NAND, 50–54, 236–237
NOR, 50–54
NOT, 23–24
OR, 22
synthesis, 40–44
XNOR, 122
XOR, 120–122
Logic gates, 7, 25–26
AND, 26, 709, 713
circuits, speed of, 714–715
drivers, 757–759
dynamic operation of, 746–749
Fall time, 748
fan-in, 755–757
fan-out, 757
NAND, 50–54, 236–237, 706–707, 712
NOR, 50–54, 708
NOT, 26, 705–706, 712
OR, 26, 710
power dissipation in, 749–751
propagation delay, 747
rise time, 747
voltage levels in, 743–744
XNOR, 122
XOR, 120–122
Logic network, 25
analysis of, 26–30
Logic values, 20
Lookup-table (LUT), 606–607, 733, 734
Loop statements. *See* FOR LOOP statement
LSB (last-significant bit), 12
LUT (lookup-table), 606–607, 733, 734

M

Macrocell, 724
Mantissa, 157
Maxterms, 48
McCluskey, Edward, 493
Mealy FSM, 310, 327–329
VHDL code, 332–334, 341–343, 696
Mealy output, 381
Mean, arithmetic, 442–445
control circuit, 445

datapath circuit, 444–445
Medium-scale integration (MSI), 720
Memory:
implied memory (VHDL), 207, 416, 674–675
static random access, 761–763
Merger diagram, 550
Merging, 548
procedure, 549–552
Metastability, 246
Metastable state
MIC (multiple-input compressor circuit), 635, 636
Minimization:
cubical technique for, 501–509
of POS form, 86–88
procedure, 83–86
of states, 351–355, 585
tabular method for, 493–501
terminology, 81–83
Minimum-cost cover, 495–500
procedure for finding, 506–508
Minterms, 44–45
Mixed logic system, 717
Moore FSM, 310
VHDL code, 692–696
Moore output, 380
Moore's law, 2–3
MOSFET transistor, 702–704
on-resistance, 742–743
Most-significant bit (MSB), 13, 124
Motherboard, 7
MSB (most-significant bit), 13
M-S D flip-flops, 240–241, 527–532
Multilevel synthesis, 466–478
Multiple faults, 619
Multiple-input compressor circuit (MIC), 635, 636
Multiple-output circuits, 90–95
Multiplexer, 56–58, 174–175
to implement bus, 402–404
VHDL code, 194, 195, 407–408
Multiplication, 152–156
algorithm for, 426
partial product, 154
of signed numbers, 154–156
of unsigned numbers, 153–154
Multiplier:
array, for unsigned numbers, 154
ASM chart for, 427
control circuit for, 428–429
datapath circuit for, 427–428
shift-and-add, 425–432

VHDL code for, 429–432
Mutual exclusion element (ME), 546

N

Named association (VHDL), 146
Names (VHDL), 650
NAND circuits, 475–478
NAND gate. *See* Logic gates
n-cube, 486
Negative edge M-S D flip-flop, 241
Negative logic system, 701, 715–717
Negative numbers, 124–128
Netlist generation, 604
Network, of gates, 25–26
Next state, 313
  variables, 313
Next-state variable, 522
Nibble, 13
9's complement, 133
NMOS transistor, 702–704
Noise, 745
  margin, 745–746
  power supply, 641
Non-disjoint decomposition, 473
Nonvolatile programming, 730
NOR circuits, 475–478
NOR gate. *See* Logic gates
NOR-NOR PLA, 765, 767
NOR plane, 765
NOT gate. *See* Logic gates
Number conversion, 118
Number display, 58–59
Number representation:
  binary-coded-decimal, 159–163
  fixed-point, 157
  floating-point, 157–159
  hexadecimal, 116–118
  9's complement, 133
  octal, 116–118
  1's complement, 126
  sign-and-magnitude, 124–126
  signed numbers, 124–138
  10's complement, 133
  2's complement, 126–128
  unsigned integer, 116
  unsigned numbers, 116, 118–124
  in VHDL code, 147–149
Numbers (in VHDL), 650

O

Octal numbers, 116–118
Odd function, 121
One-hot encoding, 185, 325, 575
1's complement
  addition, 128–129
  representation, 126
On-resistance, 742–743
ON-set, 506
#–Operation, 504–506
*–operation, 501–502
  usage to find prime implicants, 502–504
Operations. *See* Logic function
Operators (VHDL), 210–211, 656–657
Optimization. *See* Minimization
Ordering of statements (VHDL), 204, 277, 674–675
OR gate. *See* Logic gates
Oscilloscope, 643
OTHERS (VHDL), 194, 665–666
Output delay time (t_{od}), 457
Oxide capacitance, 747

P

Package (VHDL), 146–147, 660–661
Packages (physical):
  ball grid array (BGA), 733
  pin grid array (PGA), 733
  quad flat pack, 729
PAL (programmable array logic), 723–725
PAL-like blocks, 728
Parallel-access shift register, 251–252
Parallel-to-serial converter, 393
Parasitic capacitance, 746
Partial product, 154
Pass-transistor switch, 771
Path sensitizing, 621–623
PCB. *See* Printed circuit board
Physical design, 62, 609–616
Pin grid array (PGA), 733
Pins, 718
PLA (programmable logic array), 721–723, 725–727
Placement phase, 609–611
Placement process, 640
Plastic-leaded chip carrier (PLCC), 727
PLDs (programmable logic devices), 4–5, 720–721
PMOS transistors, 703–704, 741–742
p-n junction, 739
Polysilicon, 739

Port (VHDL), 66, 658
Portability, 61
PORT MAP, 662
Positional association (VHDL), 146, 662
Positional number representation, 11
Positive-edge-triggered M-S D flip-flop, 241
Positive logic system, 701, 715
Power dissipation, 749–751
  dynamic, 749
  static, 749
Power supply, 641
PRBSG (pseudorandom binary sequence generator), 635, 637
Precedence of operations, 40, 214–215
Present state, 313
  variables, 313, 522
Preset input, 243–245
Price/performance ratio, 137–138
Prime implicant, 82, 494–495
  essential, determination of, 504
  *–operation to find, 502–504
Primitive flow table, 548
Principle of duality, 31–32
Printed circuit board (PCB), 3, 640–642
  testing of, 642–643
Priority
  encoder, 190
  in VHDL code, 198, 205, 206
Processor, simple, 408–419
PROCESS statement (VHDL), 203, 671, 673–674
Process transconductance parameter, 741
Product-of-sums (POS) form, 48–50
  minimization of, 86–88
Programmable array logic (PAL), 723–727
Programmable logic array (PLA), 721–723, 725–727
Programmable logic devices (PLDs), 4–5, 720–721
  SRAM blocks in, 762–763
Programmable read-only memory (PROM), 735
Programmable switches, 4
Programmable transistor, 764
PROM (programmable read-only memory), 735
Propagation, fault, 623–625
Propagation delays, 62, 239, 748
Properties, of Boolean algebra, 32–34
Pseudo-NMOS circuits, 744
Pseudorandom binary sequence generator (PRBSG), 635, 637
Pseudorandom tests, 634
Pull-down network (PDN), 708
Pull-up network (PUN), 708
Pulse mode, 522

Q

QFP (quad flat pack) package, 729
Quine, Willard, 493
Quine-McCluskey method, 493

R

Random tests, 626–629
Reflections, 641–642
Registers, 249
  with enable inputs, 280–283
  shift. *See* Shift register
  VHDL code, 684–686
  VHDL sequential statements for, 273–283
Register transfer level (RTL) code, 290–292
Relational operators (VHDL), 212
Reliability, digital system, 643
Reset input, 232, 313
Reset state, 312
Resolution function, 652
Ring counter, 262–264
Ripple-carry adder, 122–123
Ripple counter, 253
Rise time, 747
Rotate operators (VHDL), 213–214
Routing, 611–613, 641
  channels, 732, 736
Row dominance, 497

S

Saturation region, 741
Scan-path technique, 630–633
Schematic, circuit, 25
Schematic capture tool, 60, 144
Sea-of-gates technology, 736
7-segment display, 191
  BCD-to-7-segment decoder, 191
Selected signal assignment (VHDL), 194, 666–667
Semiconductor, 739
Sensitivity list (VHDL), 203
Sequential assignment statement (VHDL), 69–70, 202–203, 669–671
Sequential circuits, 232, 310, 619, 680–696
  analysis, 376
  asynchronous. *See* Asynchronous sequential circuits
  definition of, 310
  finite state machine, 310
  formal model, 384

merger diagram, 550
state assignment, 313–315, 322–323
state diagram, 312
state minimization, 351–355, 546–559
state table, 313
synchronous, 309–397
testing of, 629–633
Serial adder, 343
Serial parity generator, 534–538
Series-to-parallel converter, 252
Setup time (t_{su}), 239
7-segment displays, 287
7400-series standard chips, 718–720
Shannon, Claude, 30
Shannon's expansion, 181–182
Shannon's expansion, derivation of BDDs using, 488–493
Shift-and-add multiplier, 425–432
Shift operators (VHDL), 213–214
Shift register, 250
with enable inputs, 283
four-bit, 274–276
parallel-access, 251–252
VHDL code, 686–688
Short-circuit current, 750
SIC (single-input compressor circuit), 635
Signal assignment statement, 67
SIGNAL data objects, 651
Sign-and-magnitude addition, 128
Sign-and-magnitude representation, 124–126
Signature analysis, 639–640
Signature pattern, 635
Signed numbers, 116, 124–138
multiplication of, 154–156
SIGNED type, 653
Simple processor, 408–419
Simple signal assignment, 664–665
Simplification. *See* Minimization
Simulation:
functional, 62
Simulator, 62
Single faults, 619
Single-input compressor circuit (SIC), 635
Single-precision floating-point format, 157–158
Single-variable theorems, 31
Slack, 614
Small-outline integrated circuit (SOIC) package, 720
Small-scale integration (SSI), 720
Socket, 727
Sort operation, 445–454
Source (in MOSFET transistor), 702

SRAM. *See* Static random access memory
Stable state, 522, 523
Standard cells, 736–738
Standard chips, 4, 718–720
Standards. *See* IEEE Standards
Starting state, 312
State, 232
  assignment. *See* State assignment
  assignment in VHDL, 338–340
  compatibility, 549
  definition of, 310
  diagram, 312
  equivalence, 351, 353
  minimization, 351–355, 546–559
  table, 313
  variables, 313
State-adjacency diagrams, 563
State-assigned table, 314, 315
State assignment, 313–315, 322–323, 560–575
  one-hot, 575
  transition diagram, 563–566
  unspecified entries, 566–569
  using additional state variables, 569–574
State machine. *See* Finite state machine
Statement ordering (VHDL), 203, 277, 674–675
Static hazards, 267, 576–581
Static power, 749
Static random access memory (SRAM), 761–763
  blocks in PLDs, 762–763
STD_LOGIC type, 652
STD_ULOGIC type, 652–653
Storage cells, 733
Storage elements
  with CAD tools, 265–273
  inclusion in schematics, 265–268
  VHDL constructs for, 268–273
Stuck-at fault model, 618
Subcircuits (VHDL), 661–663, 688–692
Substrate (in MOSFET transistor), 702
Substrate bias voltage, 752
Subtraction, 128
Subtype (VHDL), 652
Sum, 119
Sum-of-products (SOP) form, 45–47
Switch, transistor, 702–704
Switch debouncing, 459–461
Synchronous clear, 245
Synchronous clear (in VHDL), 271–273, 682–684
Synchronous counters, 254–258, 362
  with D flip-flops, 257–258

enable and clear capability, 255–256
with T flip-flops, 254–255
Synchronous reset, 260–261
Synthesis, 177–178
of asynchronous circuits, 533–546
CAD, 61–62, 604–609
logic function, 40–44
multilevel, 466–478
process, 26
System library, 660

T

Technology. *See* Integrated circuit technology
Technology mapping, 606–609
Template, gate-array, 736
10's complement, 133
Terminal nodes, 487
Terminations, 641–642
Test set, 619
complexity of, 619–621
deterministic, 626
Test/testing, 335–336, 618
of logic circuits, 618–644
T flip-flops, 247–248
down-counter with, 253–254
synchronous counter with, 254–255
up-counter with, 252–253
Theorems of Boolean Algebra, 31
Three-way light control, 55–56
Threshold voltage, 701, 740
Timing analysis, 62–63, 292–296, 613–616
with clock skew, 294–296
Timing diagram, 27, 28, 316–318
Timing issues, clock synchronization and, 454–461
Toggle, 268
Tools, CAD, 16–17, 59–60, 604–616
storage elements with, 265–273
Transistor:
EEPROM, 765–767
EPROM, 767
MOSFET, 702–704
NMOS, 702–704
PMOS, 703–704, 741–742
programmable, 764
sizes, 749
Transistor-transistor logic (TTL), 719
Transition diagram, 563–566
Transition table, 525
Transmission gates, 753–755

Transmission-line effects, 641
Tree structure, circuits with, 625–626
Triode region, 741
Tri-state:
  buffer, 400–402, 719, 759–761
Truth tables, 24–25
Turned off region, 740
2's complement
  addition, 130–131
  representation, 126–128
Type (VHDL), 332–334

U

Union, 36
Unsigned numbers, 116
  addition of, 118–124
  array multiplier for, 154
  multiplication of, 153–154
UNSIGNED type, 653
Unspecified entries, 566–569
Unstable state, 530
Up-counter, 252–253, 278–279, 414
Up/down-counter, 254
User library, 660

V

Valuation, 24
Variable assignment operator, 676
VARIABLE data objects, 655, 675–680
Variables, 20–23
Vending-machine controller, 583–588
Venn diagram, 34–38
Verilog HDL, 61
Vertices, 483
Very large scale integration (VLSI), 720
VHDL (Very High Speed Integrated Circuit Hardware Description Language), 61, 63–65, 649–699
  architecture, 66, 658–660
  arithmetic assignment, 149–152
  array, 655–656
  asynchronous clear, 271, 682
  attribute, 270, 339
  bit-counting circuit, 423–425
  BIT_VECTOR type, 651
  BUFFER, 280
  bus structures, 404–408
  CASE statement, 208, 416–419, 672–673
  comment, 650

component, 661
COMPONENT declaration statement, 146
concatenate operator, 151
concatenation, 212–213
conditional signal assignment, 161, 197–198, 667–668
constant, 650
data types, 67–69
default value, 416, 423, 674
design of arithmetic circuits using, 144–147
divider, 439–442
entity, 66, 657–660
GENERATE, 200–201, 668
hierarchical code, 70–73
IF statement, 201, 671–672
implied memory, 207, 416, 674–675
instantiation statements, 146, 661–663
libraries, 146–147, 660–661
multiplexer, 194, 407–408
named association, 146
names, 650
number representation, 147–149, 650
operators, 210–211, 656–657
ordering of statements, 204, 277, 674–675
package, 146–147, 660–661
port, 66, 658
positional association, 146
precedence, 214–215
process, 203–204
PROCESS statement, 671, 673–674
representation of digital circuits in, 65
selected signal assignment, 194, 666–667
sensitivity list, 203
sequential assignment statement, 69–70, 202–203, 669–671
sequential statements for registers and counters, 273–283
SIGNAL data objects, 651
SIGNED type, 151
subcircuits, 661–663
synchronous clear, 271–273, 682–684
VARIABLE data objects, 655, 675–680
WHILE LOOP statement, 673
Via, 736
Volatile programming, 733
Voltage levels
high, low, 701
in logic gates, 743–744
substrate bias, 752
V_{IH} and V_{IL}, 745
V_{OH} and V_{OL}, 745
Voltage transfer characteristic, 744, 745

W

WAIT UNTIL statement, 270–271, 681–682
WHEN clause (VHDL), 194, 667, 673
WHILE LOOP statement, 673

X

XNOR (Exclusive-NOR) gate. *See* Logic gates
XOR (Exclusive-OR) gate. *See* Logic gates